Proyecto de grado

Control y Comportamiento de Robots Omnidireccionales

Descripción de la Arquitectura del Sistema Sistema EasyRobots

Santiago Martínez, Rafael Sisto
pgomni@fing.edu.uy
http://www.fing.edu.uy/~pgomni

Tutor

Gonzalo Tejera

Cotutores

Facundo Benavides, Santiago Margni

Versión 2.6

Instituto de Computación
Facultad de Ingeniería - Universidad de la República
Montevideo - Uruguay

22 de noviembre de 2009

Índice

Índice de figuras

1. Introducción

1.1. Propósito

Este documento brinda una visión comprensible de la arquitectura con un enfoque orientado al desarrollo de la aplicación, así como las posibilidades de extensibilidad del mismo. Se describen las relaciones entre los componentes del sistema utilizando distintos enfoques con la finalidad de brindar una mayor comprensión del sistema a nivel interno. Se pretende que el documento brinde al lector una visión global del diseño general del framework desarrollado.

Este documento será utilizado en una próxima instancia por el equipo de desarrollo del sistema, y a posteriori con el fin de extender el mismo en alguno de sus aspectos.

1.2. Alcance

El documento se centra en el desarrollo de la vista lógica del *framework*. Se incluyen los aspectos fundamentales del resto de las vistas y se omiten aquellas que no se consideren relacionadas al sistema a desarrollar.

1.3. Definiciones, Acrónimos y Abreviaciones

Dirección MAC la dirección MAC (Media Access Control address o dirección de control de acceso al medio) es un identificador que corresponde de forma única a una tarjeta o interfaz de red.

Framework Representa una arquitectura de software que modela las relaciones generales de las entidades de dominio, además de proveer una estructura y una metodologia de trabajo la cual extiende o utiliza las aplicaciones de dominio.

LAN Red de area local (Del inglés Local Area Network)

RUP Rational Unified Process. Modelo de proceso para desarrollo de un sistema informático desarrollado por la empresa Rational de IBM.

XML siglas en inglés de Extensible Markup Language (lenguaje extensible de marcas), es un metalenguaje extensible de etiquetas desarrollado por el World Wide Web Consortium (W3C).

1.4. Organización del Documento

El documento se desarrolla y organiza en base a la plantilla elaborada para el artefacto Software Architecture Document del proceso de desarrollo de software elaborado por RUP[1, 2], adaptada a las características particulares del tipo de proyecto en desarrollo. En la sección 2 se da una descripción de las distintas vistas que serán desarrolladas en las siguientes secciones.

Las siguientes secciones se abocan a la descripción de la arquitectura del sistema EasyRobots, entrando en detalle de las vistas definidas.

2. Representación de la arquitectura

El modelo propuesto por RUP para representar la arquitectura utiliza el siguiente conjunto de vistas:

- Vista de Casos de Uso: lista los casos de uso o escenarios del modelo de casos de uso que representen funcionalidades centrales del sistema final, que requieran una gran cobertura arquitectónica o aquellos que impliquen algún punto especialmente delicado de la arquitectura.
- Vista Lógica: describe las partes arquitectónicamente significativas del modelo de diseño, como ser la descomposición en capas, subsistemas o paquetes. Una vez presentadas estas unidades lógicas principales, se profundiza en ellas hasta el nivel que se considere adecuado.
- Vista de Procesos: describe la descomposición del sistema en threads y procesos pesados. Indica que procesos o grupos de procesos se comunican o interactúan entres sí y los modos en que estos se comunican.
- Vista de Distribución: describe uno o más escenarios de distribución física del sistema sobre los cuales se ejecutará y desplegará del mismo. Muestra la comunicación entre los diferentes nodos que componen los escenarios antes mencionados, así como el mapeo de los elementos de la Vista de Procesos en dichos nodos.
- Vista de Implementación: describe la estructura general del Modelo de Implementación y el mapeo de los subsistemas, paquetes y clases de la Vista Lógica a subsistemas y componentes de implementación.
- Vista de Datos: describe los elementos principales del Modelo de Datos, brindando un panorama general de dicho modelo en términos de tablas, vistas, índices, etc.

3. Vista de Casos de Uso

En esta sección se describen los Casos de Uso relevantes para definir la arquitectura del sistema a desarrollar. Se detalla más información sobre estos casos de uso en el documento de Modelo de Casos de Uso[3].

También se considerarán aquellos requisitos funcionales como no funcionales definidos en el documento Especificación de Requerimientos de Software[4].

En la figura 1 se presenta un diagrama con los Casos de Uso identificados como relevantes para la arquitectura. A continuación se detallarán éstos y se explicará brevemente el motivo por el cual fueron incluidos.

3.1. Inicialización del sistema

El usuario inicializa el sistema y éste crea las instancias que reflejen el estado inicial solicitado. Para ello el usuario debe brindar un archivo que describa una aplicación robótica conteniendo uno o varios comportamientos que se podrán utilizar, los dispositivos de visión que serán utilizados por el sistema, la especificación de la estructura de los robots, así como otros atributos necesarios para definir completamente la aplicación.

Este caso de uso es relevante debido a que cada entidad definida en la aplicación robótica debe estar declarada en un archivo del sistema con cierta estructura definida por la entidad lógica del sistema. Dada la precondición de que el sistema debe ser extensible de forma de incluir nuevas entidades (robots, sistemas de posicionamiento, comportamientos, planificación de trayectorias, etc.) cada entidad lógica debe tener su cargador específico.

3.2. Activación de Comportamientos

El usuario activa uno de los comportamientos definidos que controlará a los robots identificados en el.

Este caso de uso es relevante para la arquitectura debido a que determina la necesidad de un punto de entrada de datos por parte del usuario al sistema en tiempo de ejecución. Los casos de uso de este estilo definirán que interfaces debe tener la capa lógica del sistema.

3.3. Recepción de datos de posicionamiento

El sistema recibe mensajes de las Aplicaciones Sensoriales conteniendo información sobre la posición o velocidad de uno o más robots. El mensaje está compuesto por los identificadores de los robots involucrados y sus posiciones o velocidades actuales. Posteriormente, el sistema procesa dichos datos obteniendo las velocidades de cada robot para ejecutar la acción que mejor se consideró de acuerdo al comportamiento utilizado. Estas velocidades se envían a los sistemas de actuación que se encargarán de enviarlas a los robots correspondientes.

Este caso de uso es relevante para la arquitectura debido a que se debe proveer una interfaz genérica para la recepción de datos brindados por los distintos sistemas de sensado y otra para la comunicación con los distintos sistemas de actuación.

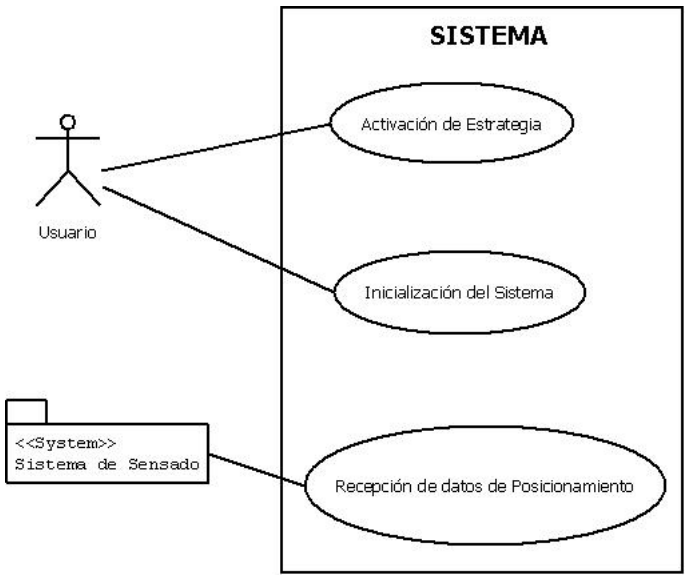

Figura 1: Diagrama de los Casos de Uso relevantes para la arquitectura.

4. Vista Lógica

La presente vista lógica describe el sistema en varios niveles de abstracción, partiendo del nivel más abstracto agregando mayor granularidad en los posteriores diagramas.

Se intenta asignar responsabilidades específicas a cada paquete y subsistema, de manera de maximizar la cohesión y minimizar el acoplamiento, según las recomendaciones de buenas prácticas de diseño en la Ingeniería de Software[5].

El primer refinamiento realizado consiste en la descomposición del sistema en los distintos subsistemas. Estos subsistemas consisten en un agrupamiento de funcionalidades específicas relacionadas de forma tal que que al integrarse, funcionan de forma cohesiva de acuerdo a los requerimientos definidos para el sistema.

Posteriormente se describe la composición de cada uno de estos subsistemas internamente y finalmente se brinda el diseño detallado de cada uno de estos paquetes.

4.1. Descomposición en Subsistemas

La figura 2 muestra la descomposición en subsistemas del sistema EasyRobots. La descomposición propuesta se basa en el patrón Model View Controller [6] para la interacción entre la interfaz gráfica y el modelo.

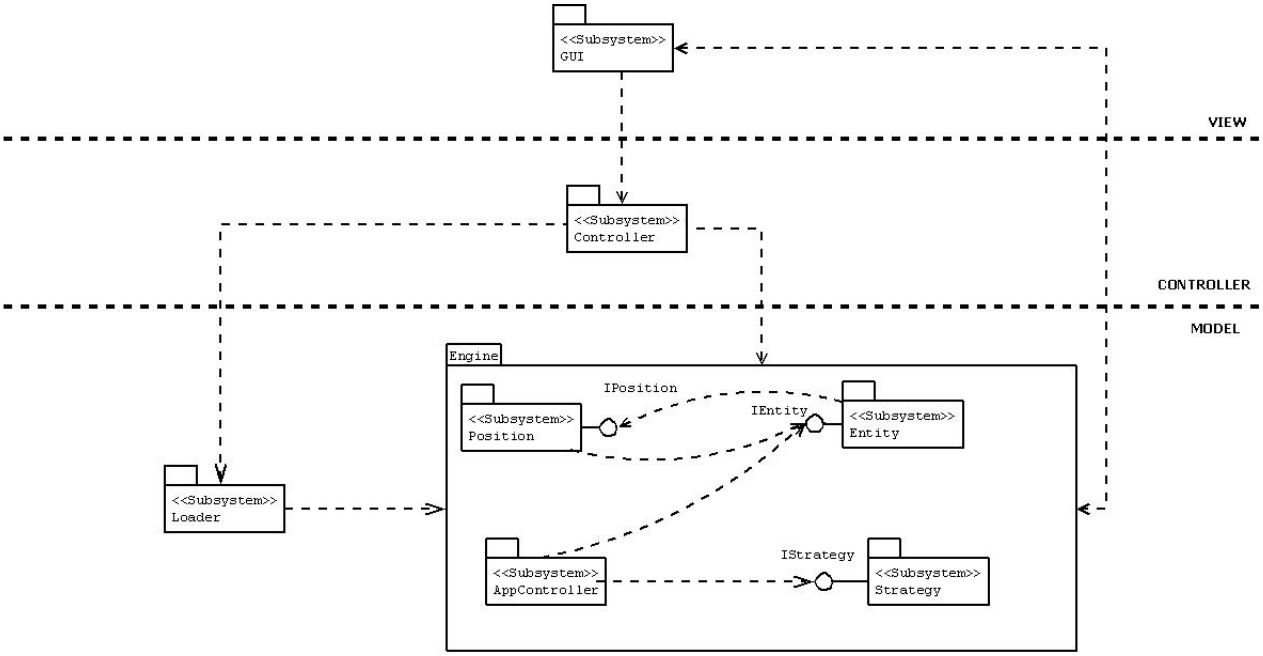

Figura 2: Arquitectura de la aplicación

4.2. Descripción de Subsistemas

4.2.1. GUI

Este subsistema representa la interfaz de usuario desde la cual se interactuará con el sistema. A partir de ésta se podrá activar un comportamiento, cargar una nueva aplicación róbotica, así como desplegar información del estado de los sistemas mediante notificaciones recibidas de éstos. Para recibir notificaciones de cambios de los subsistemas *Position* y *Entity*, la interfaz se deberá registrar como observador a cada subsistema (ver en 4.3.4 y 4.3.5 que dichos subsistemas permiten que observadores se puedan registrar).

4.2.2. Controller

Este subsistema será el encargado de comunicar la interfaz gráfica con la capa lógica del sistema. Su objetivo es el de aislar la solicitud y procesamiento de las solicitudes del usuario a través de su intefaz de la capa lógica. Para ello, recibirá pedidos desde la interfaz de usuario y éste los procesará e invocará a las operaciones correspondientes de la capa lógica.

4.2.3. Loader

Este subsistema es el encargado de instanciar una aplicación robótica a partir de un archivo XML. En este archivo se encuentran definidas las entidades del sistema; el subsistema creará las instancias necesarias para la posterior ejecución de la aplicación robótica.

4.2.4. Position

Este subsistema es el encargado de manejar las distintas fuentes de datos de posicionamiento de robots y entidades. En éste se cargarán los distintos Sistemas de Sensado y se podrán procesar mediante algoritmos de *Sensor Data Fusion*. En este sentido, notifica cambios en las mediciones de los sensores a otros componentes del sistema para que dichos datos sean consultados y procesados.

4.2.5. Entity

Este subsistema maneja todos los robots y entidades de la aplicación robótica y maneja la comunicación física con el robot. Adicionalmente traduce las velocidades en el plano a velocidades de ruedas para los robots definidos. Es el encargado de mantener y centralizar los datos de los robots y entidades, es decir, sus posiciones, velocidades, estructuras, entre otras.

4.2.6. AppController

Este subsistema es el encargado de ejecutar la aplicación robótica. Para ello, toma datos de los robots y entidades, y aplica un comportamiento sobre éstos. Finalmente notifica a los robots sus nuevas velocidades en el plano.

4.2.7. Strategy

Este subsistema mantiene los comportamientos definidos por el usuario.

4.3. Diseño de Subsistemas

En esta sección se diseñará en mayor detalle los subsistemas descritos en la sección anterior.

4.3.1. Subsistema GUI

Este subsistema no presenta un diseño complejo y al tratarse de una interfaz de usuario se deja a criterio del programador que clases implementar según que plataforma gráfica se desea desarrollar. Para comunicarse con el resto del sistema se deberán utilizar las interfaces brindadas por el subsistema *Controller* definida en la sección 4.3.2, aquella definida por el subsistema Position 4.3.4y por el subsistema Entity 4.3.5.

4.3.2. Subsistema Controller

Este subsistema es utilizado como acceso a la capa lógica desde la interfaz de usuario (Subsistema *GUI*). Se brinda la interfaz *IController* para proveer los métodos necesarios a la interfaz de usuario.

Las componentes de este sistema son las siguientes:

- IController: interfaz implementada por el controlador del patrón Model View Controller utilizado en 4.1. Las funciones de esta clase son:
  - loadApplication: esta función es utilizada para cargar la aplicación especificada por el archivo que se encuentra en la ruta brindada por el parámetro *filePath*.
  - activateStrategy: esta función es utilizada para activar un comportamiento indicado en el parámetro *idStrategy*.
  - deactivateStrategy: esta función es utilizada para desactivar un comportamiento indicado en el parámetro *idStrategy*.

Los métodos provistos por el subsistema *Controller* se definen en la interfaz que se puede ver en la figura 3.

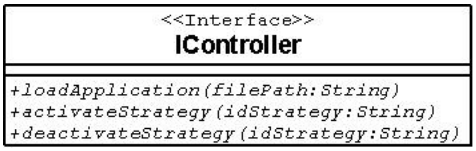

Figura 3: Interfaz brindada por el subsistema *Controller*

El diseño detallado del subsistema se puede ver en la figura 4. Las nuevas componentes detalladas de este sistema son:

- Controller: Implementa la interfaz IController. Se diseña a través de la utilización de los patrones Facade y Singleton [6]. El patrón Singleton se utiliza de forma de tener una única instancia de este objeto, ya que no se requieren más puntos de entrada al sistema desde la interfaz de usuario. El patron Facade se utiliza debido a que *Controller* tiene métodos que son derivados a distintos subsistemas, según cual sea el encargado de manejar el pedido.

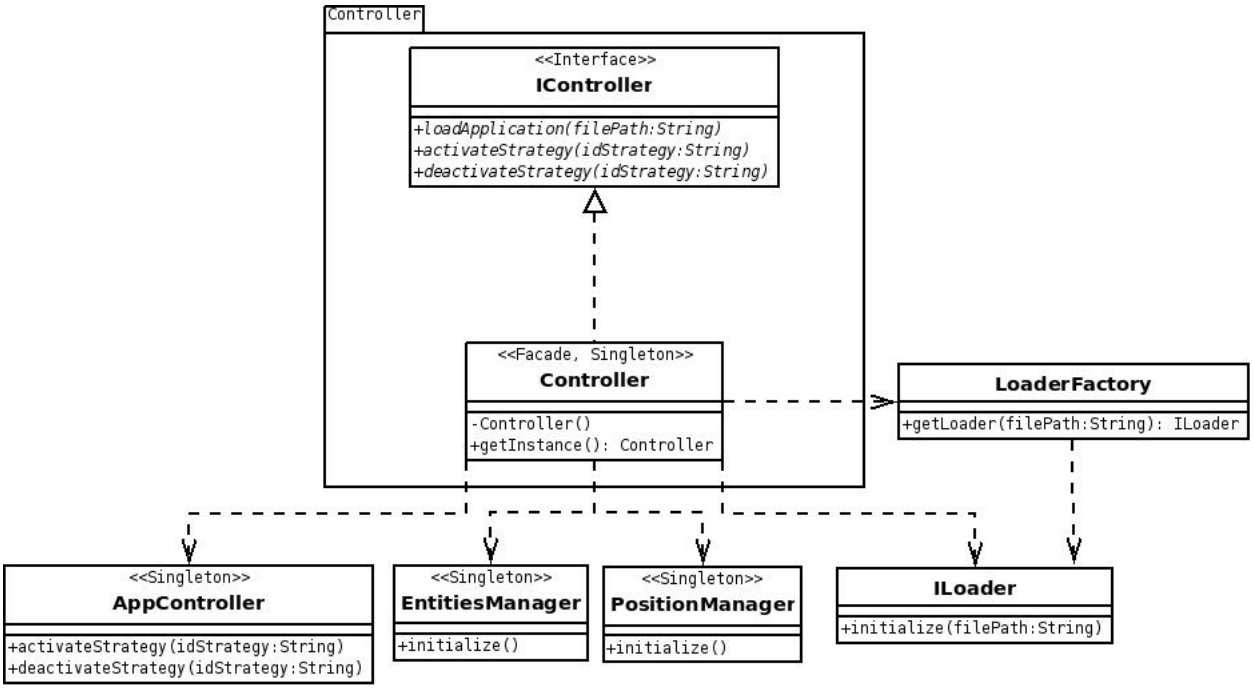

Figura 4: Diseño detallado del subsistema Controller

Funcionamiento En esta sección se describe a grandes rasgos el funcionamiento de este subsistema. Los detalles específicos se libran a criterio del programador, mientras se respeten los criterios aquí definidos.

El funcionamiento general consiste en la carga de una aplicación robótica al sistema, así como la activación y desactivación de comportamientos.

Función loadApplication Cuando se desea cargar una nueva aplicación de usuario, el controlador delega el trabajo al subsistema *Loader* utilizando la interfaz brindada por el mismo (definida en la sección 4.3.3), en particular invocando a la función initialize. Para obtener una instancia específica de ILoader, se le solicita a LoaderFactory utilizando como parámetro el mismo parámetro brindado por la interfaz de usuario.

Al finalizar correctamente la invocación del método, quedará cargada en el sistema una aplicación robótica definida en el archivo solicitado. Al finalizar la carga, se invoca a la función initialize de los manejadores de Robots y de Sistemas de Sensado a fin de que se construyan todas las referencias entre estos tipos de entidades.

Funciones activate/deactivateStrategy Cuando se desea activar o desactivar un comportamiento cargado en el sistema, el controlador delega el trabajo al subsistema *AppController*, invocando a las funciones activateStrategy y deactivateStretegy según corresponda. La descripción de éstas está definidas en la sección 4.3.7.

4.3.3. Subsistema Loader

Este subsistema es el utilizado para, a partir de un archivo en un formato específico, inicializar el sistema de forma que represente la aplicación robótica. La figura 5 muestra el diseño detallado de la aplicación.

Los componentes de este subsistema son las siguientes:

- ILoader: interfaz implementada por aquellas clases encargadas de cargar la aplicación robótica a partir de un formato de archivo específico. Las funciones de esta clase son:
  - initialize: esta función se utiliza para comenzar a cargar la aplicación a partir de un archivo especificando su ruta en el parámetro filePath.
- LoaderFactory: esta clase se encarga de instanciar un cargador específico para un tipo de archivos conteniendo una aplicación robótica
  - getLoader: a partir de una ruta a un archivo conteniendo una aplicación robótica, devuelve el ILoader encargado de cargar ese tipo de archivos.
- XMLLoader: clase encargada de cargar la aplicación a partir de un archivo en formato XML.
- XMLSubLoaderFactory: clase encargada de instanciar a las clases cargadoras durante la inicialización de la aplicación a partir de un archivo XML. Esta clase se diseña utilizando el patrón Factory y Singleton [6]. El patrón Factory se utiliza para que a partir de un identificador de tipo de clase se instancie un cargador para la clase identificada. El patrón Singleton se utiliza para asegurar que exista una única instancia que permita obtener los diferentes cargadores. Las funciones de esta clase son:
  - getSpecificLoader: esta función recibe como parámetro el identificador de una clase e instancia el cargador que es capaz de cargar la información del archivo de entrada para la parte correspondiente, en dicho archivo, a la información de la clase especificada por el identificador.
- IXMLSubLoader: interfaz implementada por los cargadores de los diferentes tipos de componentes de la aplicación. Las funciones de esta interfaz son:
  - parseEntry: esta función permite a partir de un texto crear instancias de las clases que corresponda según lo especifique la descripción del texto.
- RobotLoader: clase encargada de cargar instancias de IRobot (4.3.5) a partir de un bloque de texto que especifique las características (valores de los atributos) de dicha clase. Las funciones de esta clase son:
  - parseEntry: esta función utiliza un texto que es recibido como parámetro y crea una instancia de IRobot asignando los valores a los atributos y creando otras instancias necesarias.
- PositionLoader: clase encargada de cargar instancias de *IPosition* (4.3.4) a partir de un bloque de texto que especifique las características (valores de los atributos) de dicha clase. Las funciones de esta clase son:
  - parseEntry: esta función utiliza un texto que es recibido como parámetro y crea una instancia de *IPosition* asignando los valores a los atributos y creando otras instancias necesarias.
- StrategyLoader: clase encargada de cargar instancias de IStrategy (4.3.6) a partir de un bloque de texto que especifique las características (valores de los atributos) de dicha clase. Las funciones de esta clase son:
  - parseEntry: esta función utiliza un texto que es recibido como parámetro y crea una instancia de IStrategy asignando los valores a los atributos y creando otras instancias necesarias.

- ManagerFactory: clase encargada de instanciar a los manejadores; cada uno es responsable de un tipo de clase diferente durante la inicialización de la aplicación. Esta clase es diseñada utilizando el patrón Factory y Singleton [6]. El patrón Factory se utiliza ya que a partir de un tipo de clase específico la clase debe instanciar a un manejador particular. El patrón Singleton se utiliza para asegurar que exista una única instancia que permita obtener las instancias de los diferentes manejadores. Las funciones de esta clase son:

  - getManager: esta función es utilizada para instanciar un manejador a partir de la instancia de una clase particular recibida en el parámetro classInstance.

- IManager: interfaz implementada por los manejadores de los diferentes tipos de componentes de la aplicación. Las funciones de esta clase son:

  - addEntity: esta función es la utilizada para agregar una entidad al manejador que particular. La entidad es recibida en el parámetro entity.

- RobotManager: (Pertenece al subsistema Entity) manejador encargado de contener las instancias de robots de la aplicación robótica. El diseño se realiza utilizando el patrón Singleton [6] para asegurar que el manejo de dichas instancias se encuentre centralizado. Las funciones de esta clase son:

  - addEntity: esta función se utiliza para agregar una instancia que representa a un robot a la colección de robots mantenidos. Dicha instancia es recibida en el parámetro robotEntity.

- PositionManager: (Pertenece al subsistema *Position*) manejador encargado de contener las instancias de los Sistemas de Sensado de la aplicación robótica. El diseño se realiza utilizando el patrón Singleton [6] para asegurar que el manejo de dichas instancias se encuentre centralizado. Las funciones de esta clase son:

  - addEntity: esta función se utiliza para agregar una instancia que representa a un Sistema de Sensado a la colección de sistemas controlados. Dicha instancia es recibida en el parámetro positionEntity.

- StrategyManager: (Pertenece al subsistema Strategy) manejador encargado de contener las instancias de los comportamientos presentes en la aplicación robótica. El diseño se realiza utilizando el patrón Singleton [6] para asegurar que el manejo de dichas instancias se encuentre centralizado. Las funciones de esta clase son:

  - addEntity: esta función se utiliza para agregar una instancia que representa un comportamiento a la colección de comportamientos controlados. Dicha instancia es recibida en el parámetro *strategyEntity*.

Funcionamiento El funcionamiento de este subsistema corresponde a la inicialización estática de las entidades del sistema. Se define el término inicialización estática como la creación de las entidades que participan en la aplicación pero sin crear las relaciones entre éstas. Esta decisión se consideró para evitar problemas de referencias circulares entre las instancias, ya que algunas entidades pueden ser observadores y necesitan registrarse a alguna entidad todavia no cargada.

Función initialize El funcionamiento de la inicialización consiste en la creación de las instancias de las entidades que participan en el sistema. Para ello se invoca a la función initialize del ILoader que maneja el tipo de archivo que se utilizará. En esta función, se abrirá el archivo que especifíca la aplicación robótica y se comenzará a leerlo de forma de ir obteniendo las distintas componentes. Para cada una de ellas, se le solicitará a la clase XMLSubLoaderFactory que retorne el loader específico para la componente que se está cargando. A continuación, se le solicita a este loader específico que cargue la componente. A continuación se instanciará a la clase ManagerFactory para invocar la función getManager con la instancia de la clase específica creada. La función getManager retornará el manejador específico para la instancia solicitada. Luego, el loader le indicará a dicho manejador que registre la instancia creada de la clase leída del archivo.

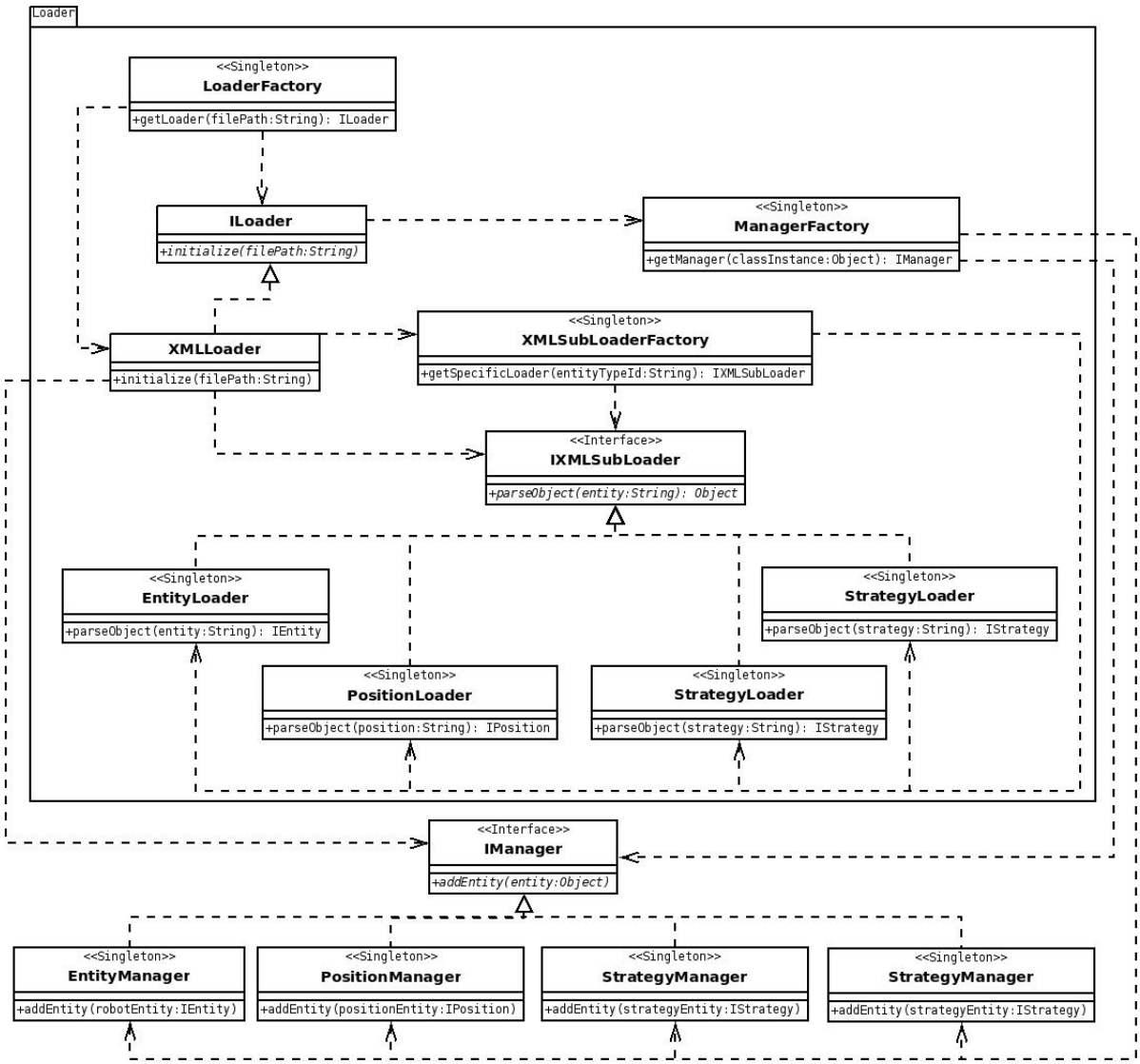

Figura 5: Diseño del subsistema Loader

4.3.4. Subsistema Position

Este subsistema representa una estructura genérica para manejar datos de posicionamiento de objetos. Está diseñado de forma tal que se pueda extender incluyendo otros Sistemas de Sensado. En la figura 6 se puede ver el diseño detallado de este subsistema.

Las interfaces brindadas por este subsistema son:

- IPosition: interfaz que representa una entidad de posicionamiento. Ésta es implementada por aquellas clases encargadas de obtener datos de posicionamiento de las entidades del sistema. Esta clase extiende la interfaz Observable, siguiendo la definición del patrón Observer[6], la cual utilizarán los demás subsistemas para recibir actualizaciones de cambios en la posición de las entidades. Las funciones de esta clase son:

  - getPosition: es utilizada para obtener la posición actual de una entidad del sistema.
  - Métodos específicos de la interfaz Observer:
    - registerObserver: agrega un Observer que será notificado ante un cambio.
    - unregisterObserver: quita un Observer de la lista de objetos a ser notificados ante cambios.
    - notifyObservers: es llamada cuando la posición cambia; notifica a los observadores.

- initialize: sirve para inicializar un sistema de posicionamiento. Una vez cargado el sistema, se llama este método de modo que la implementación específica de esta clase genere los vínculos que se necesiten para el correcto funcionamiento.

- IManager: interfaz implementada por los manejadores de los diferentes tipos de componentes de la aplicación. Las funciones de esta clase son:

  - addEntity: esta función es la utilizada para agregar una entidad al manejador particular.

Los componentes de este subsitema son los siguientes:

- AbstractSensor: clase abstracta que implementa *IPosition*. Implementa los métodos específicos de Observer, y manejo de posiciones básico. Las funciones de esta clase son:

  - getConfidence: función utilizada internamente para describir la confianza del sensor. Se define esta función para poder ponderar las mediciones realizadas de acuerdo a la incertidumbre de los sensores.
  - setPosition: actualiza la posición de una entidad física y notifica a los observadores.

- SensorFusion: clase abstracta representando el tipo de posicionamiento utilizando Sensor Data Fusion. Se utiliza para representar la idea principal de fusión de datos con ayuda del patrón de diseño Composite[6]. Un método de Sensor Fusion está compuesto por varios sistemas de posicionamiento genéricos (pudiendo ser otro método de Sensor Fusion). El atributo idEntity determina para que entidad se está realizando la fusión de sensores. Las funciones de esta clase son:

  - run: procesamiento general de los algoritmos de fusión de sensores, se implementa mediante el patrón de diseño Template Method[6], donde el método abstracto es doFusion, el cual deberán implementar las subclases específicas.
    El funcionamiento es el siguiente: cada un cierto período de tiempo definido, se calcula la nueva posición fusionando los datos de sensores asociados y finalmente se actualiza la nueva posición de la entidad.
  - doFusion: esta función implementa la fusión de datos específica . Esta función es abstracta y será implementada por las subclases, como pueden ser AverageSensorFusion u OtherSensorFusion.

- GlobalPositionSensor: clase representando los sistemas de posicionamiento global, como pueden ser sistemas de posicionamiento por triangulación de sensores, ó cámaras de vista superior.
  Las subclases de ésta dan libertad para agregar métodos y atributos necesarios para el funcionamiento del sistema de posicionamiento que se desee, como se muestra el ejemplo de la clase DoraemonClient.

- PredictionSensor: esta clase representa los sistemas de posicionamiento basados en la predicción de la nueva posición de un objeto utilizando un sistema de posicionamiento como referencia y las velocidades del objeto para determinar la nueva posición. Esta clase actualiza la posición de la entidad cada cierto período de tiempo utilizando el sensor de referencia y la velocidad conocida de la entidad.
  Esta clase implementa el rol de observador de acuerdo al patrón *Observer*[6], utilizando como rol observable a una instancia de la clase *IEntity*. Este patrón se utiliza para registrarse a una entidad (*IEntity*) de forma de recibir notificaciones de cambios de velocidad.
  Las funciones de esta clase son:

  - update: método del patrón *Observer*, en este método se deberá actualizar la posición de la entidad de modo de ajustarse a la nueva posición de la misma.

- PositionManager: manejador de los sistemas de posicionamiento. En esta clase se registrarán todos los sistemas de posicionamiento que estén ejecutándose en el sistema. A través de este se puede acceder a un sistema de posicionamiento utilizando un identificador.

Funcionamiento Este subsistema, una vez cargado de la persistencia e inicializado, funciona de forma autónoma, manteniendo almacenadas en memoria las posiciones de todas las entidades del sistema.

Al detectar cambios en los datos del posicionamiento de alguna entidad, se notificará a los observadores de la posición de ésta, en caso de existir alguno.

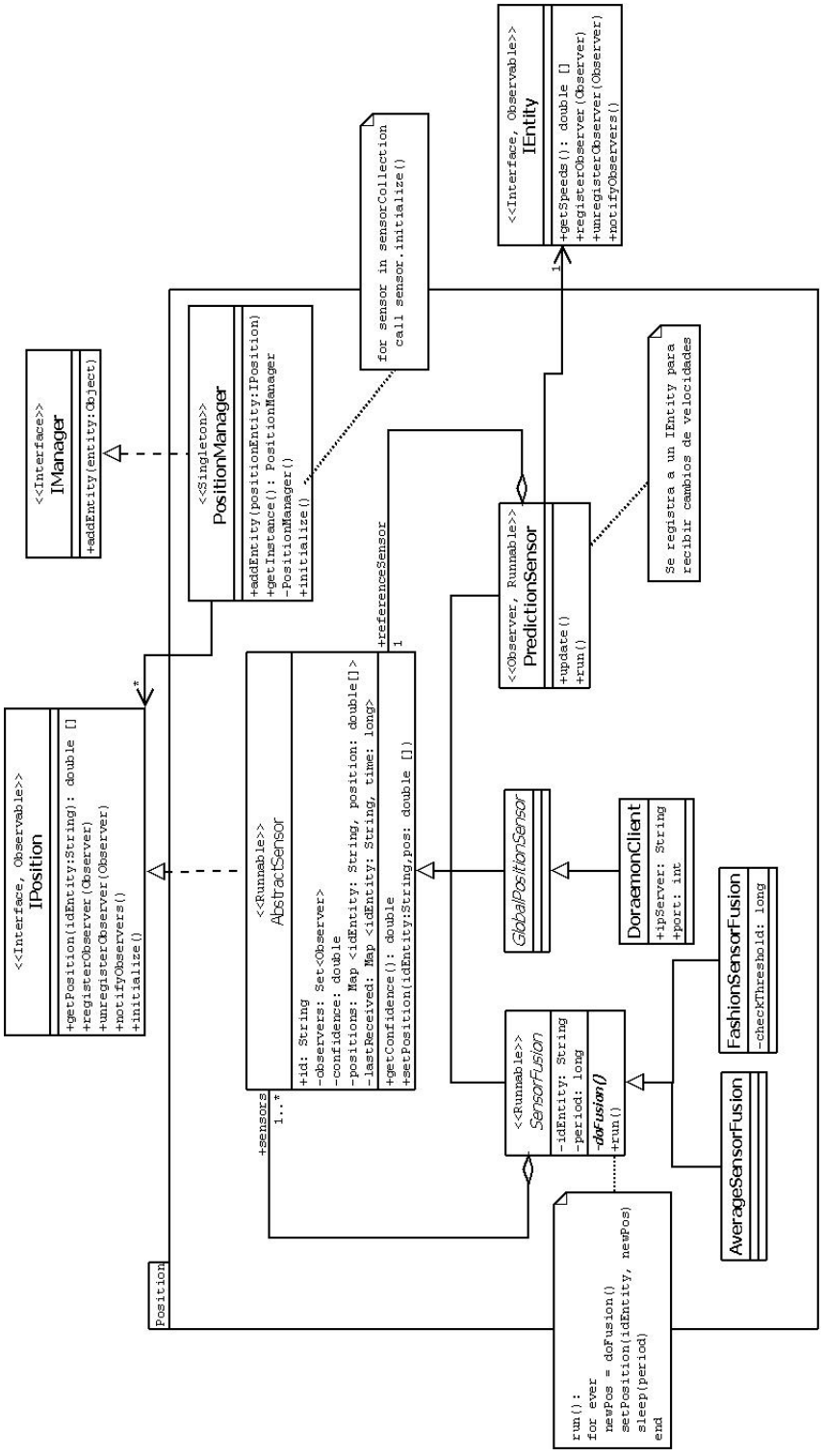

Figura 6: Diseño detallado del subsistema *Position*

4.3.5. Subsistema Entity

Este subsistema es el utilizado para representar una entidad física dentro de la aplicación. Se identifica la subdivisión de las clases del subsistema en dos paquetes componentes: *LogicalEntities* y *PhysicalRobots*. La figura 7 muestra esta división.

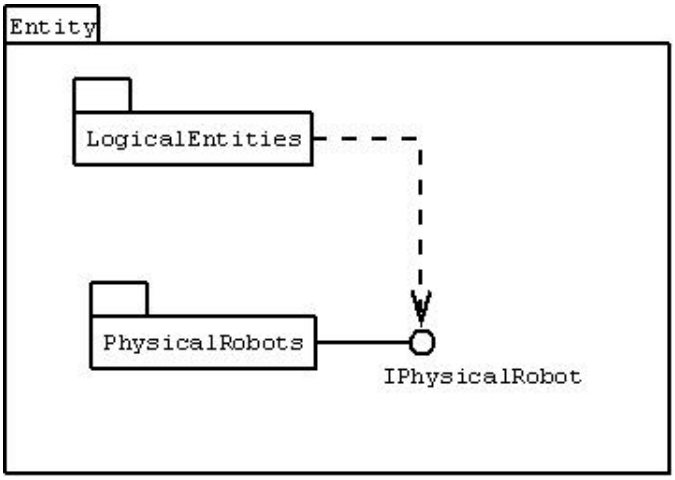

Figura 7: Componentes del subsistema Entity

El paquete *LogicalEntities* modela a las entidades (robots, obstáculos, pelota) de forma lógica aislando la lógica de la comunicación con los robots físicos. En este paquete se guardan los datos de las entidades como son:

- los datos en tiempo de ejecución (posición y velocidad),
- la estructura (tamaño, distribución de ruedas para los robots, etc) y
- para los robots controlados por el sistema: algoritmos de traducción de velocidades en el plano a velocidades en las ruedas, para su posterior envío a los robots físicos (a través del paquete *PhysicalRobots*).

El diseño específico de este paquete se puede observar en la figura 8.
Se brinda la interfaz *IEntity* para obtener los datos relevantes para los otros subsistemas.
Las clases que se encuentran en el paquete son:

- IEntity: intefaz implementada por las entidades fisicas de la aplicación robótica. A fin de retroalimentar a un posible conjunto de sensores, se diseñó haciendo uso del patrón Observer. Las funciones de ésta son:
  - getSpeeds: retorna la velocidad de la entidad
  - getPosition: retorna la posición de la entidad
  - registerObserver: registra un observador para ser notificado al producirse un cambio en su velocidad.
  - unregisterObserver: quita a un observador de la lista de observadores de su velocidad.
  - notifyObservers: envía una notificación a sus observadores sobre un cambio en su velocidad.
  - initialize: crea las relaciones entre las clases al momento de inicializar el sistema.
- IManager: interfaz implementada por los diferentes manejadores del sistema, en este caso por el manejador EntityManager. Las funciones de esta interfaz son:
  - addEntity: agrega una entidad al manejador.
- EntityManager: clase que implementa el manejador de entidades a fin de agrupar todas las entidades del sistema. Se diseña siguiendo el patrón Singleton a fin de que se cuente con una única instancia en el sistema. Cuenta con la función:
  - initialize: inicializa todas las entidades *IEntity*.

- AbstractEntity: representa una entidad física genérica de la aplicación robótica. Se diseña siguiendo el patrón Observer a fin de registrarse a los cambios de posición que notifique el subsistema *IPosition* 4.3.4. Cuenta con las siguientes funciones:

  - update: es invocada desde el sistema *IPosition* a fin de notificar sobre el cambio de la posición de la entidad.

- IRobot: interfaz que extiende a la interfaz *IEntity* de forma de brindar funciones adicionales para los casos de que la entidad sea un robot. Las funciones de esta clase son:

  - setSpeeds: modifica la velocidad del robot físico. es invocada por el subsistema *AppController* (sección 4.3.7) cuando se desea enviar un comando de velocidades a los robots.

- AbstractRobot: representa a un robot genérico. Esta clase es extendida por los distintos tipos de robots, ya que éstos necesitarán implementar distintos algoritmos de traducción de velocidad en el plano a velocidad en las ruedas, ya que el algoritmo es distinto para robots omnidireccionales y otros. Las funciones de esta clase son:

  - setSpeeds: envía la nueva velocidad para que sea procesada por la contraparte de este robot en el paquete *PhysicalRobot*.

- OmnidirectionalRobot: clase que representa a un robot omnidireccional. Posee instancias de la clase Wheel para representar su estructura física.

- BiWheeledRobot: clase que representa a un robot de dos ruedas.

- Wheel: clase que representa una rueda en un robot. Posee atributos que determinan su posición en el robot así como sus características específicas detalladas en [7].

Funcionamiento Este componente tiene como cometido mantener la representación lógica de las entidades. Para ello, se manejan las velocidades y posiciones de éstas recibiendo datos de las posiciones desde el subsistema *IPosition* y de las velocidades desde el subsistema *AppController*. En este paquete se calculan las velocidades a asignar a cada rueda a partir de las velocidades deseadas en el plano; a continuación son enviadas al componente *PhysicalRobots* para que ésta las envíe a los robots físicos. Se explicará el funcionamiento de aquellas funciones que se consideren complejas y de mayor trascendencia para el correcto funcionamiento.

Función update de la clase AbstractEntity Esta función es invocada desde el subsistema *Position* a fin de notificar sobre el cambio en la posición de la entidad. En esa notificación, se solicita la nueva posición al subsistema *Position* a través de la función *getPosition* de la interfaz *IPosition*.

Función setSpeeds de la clase AbstractRobot Esta función es invocada desde el subsistema *AppController* para modificar las velocidades de los robots. En ella, se calculan las velocidades a asignar a cada rueda a través de la función *getWheelSpeeds* y el resultado es enviado al componente *PhysicalRobots* mediante la función *setWheelSpeeds*.

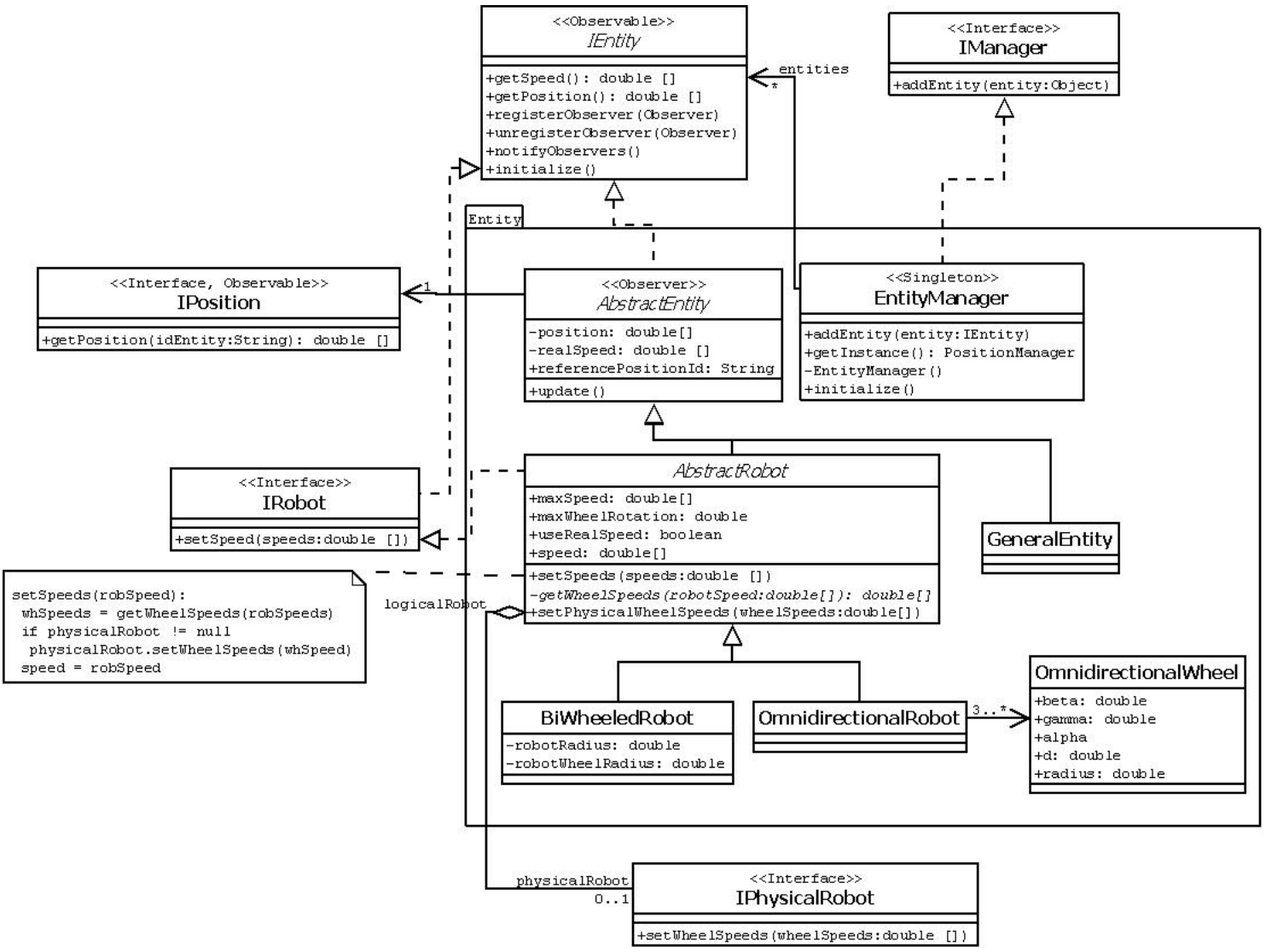

Figura 8: Diseño del paquete LogicalEntities.

El paquete *PhysicalRobots* modela la comunicación del sistema con el robot. La figura 9 muestra las clases que se encuentran en el paquete. Como punto de entrada al sistema, se encuentra a la interfaz *IPhysicalRobot* que provee las funciones necesarias para los otros subsistemas. Las clases que se encuentran en el paquete son:

- IPhysicalRobot: interfaz que funciona como punto de entrada al sistema. Posee las siguientes funciones:
  - setWheelSpeeds: esta función es utilizada por el subsistema *LogicalEntities* a fin de comunicar las nuevas velocidades al robot físico.
- PhysicalRobot: clase que implementa a la interfaz *IPhysicalRobot*. Posee un único atributo (*robotSpeeds*) que guarda las velocidades de las ruedas del robot.
- NXTRobot: clase que extiende a *PhysicalRobot* de forma de representar un robot NXT[8]. Posee el atributo id que indica el identificador del robot. Este atributo (dirección MAC del robot) es utilizado al momento de construir el mensaje con las velocidades de cada robot que será enviado a los robots.
- OtherPhysicalRobot: clase que ejemplifica que otros robots pueden ser implementados, simplemente extendiendo a la clase PhysicalRobot.

- NXTController: clase que maneja el conjunto de robots de tipo NXT presentes en la aplicación robótica. Posee los atributos necesarios (*ipServer* y *portServer*) para indicar la ubicación de red del servidor de comunicación con los robots.

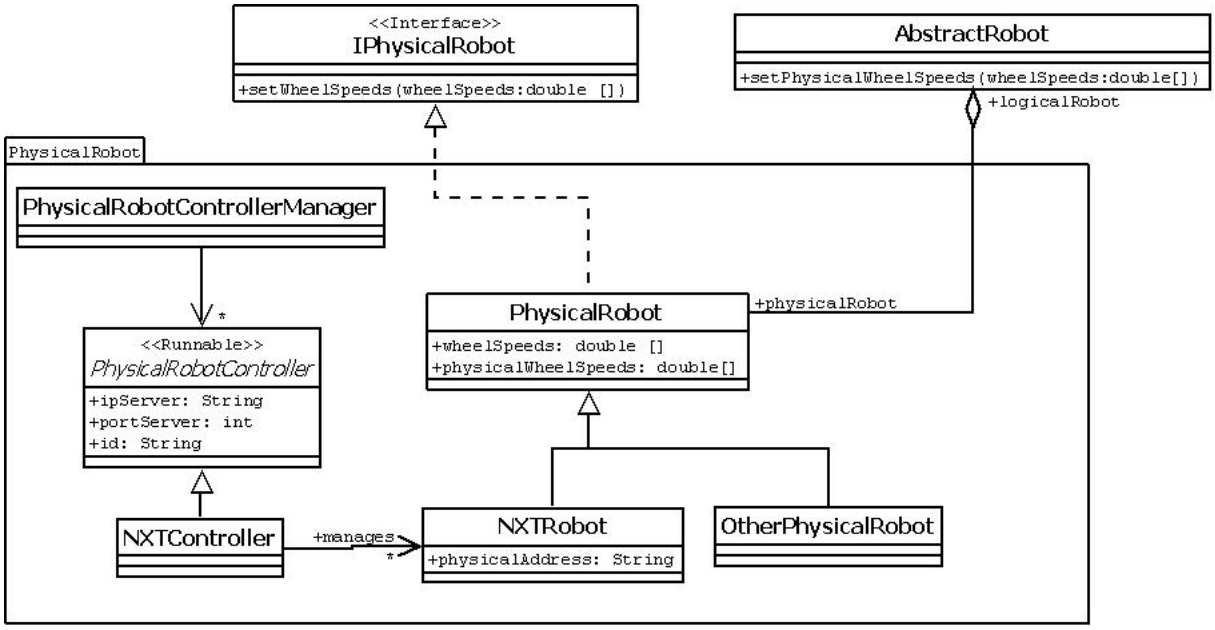

Figura 9: Diseño del paquete *PhysicalRobots*

Funcionamiento La función setWheelSpeeds de la interfaz IPhysicalRobot es invocada desde el paquete *Logical-Entites* a fin de modificar la velocidad de cada rueda de un robot. Este cambio se refleja en el cambio del atributo robotSpeeds de la clase PhysicalRobot. La clase NXTController, cada cierto tiempo solicita dicho dato y lo envía a los robots a través del medio de comunicación que se utilice, por ejemplo un servidor de comunicación.

4.3.6. Subsistema Strategy

Este subsistema mantiene los comportamientos definidos en la aplicación robótica. Serán accedidos desde el subsistema *AppController* 4.3.7. Este sistema será utilizado por la aplicación robótica para calcular las velocidades de los robots luego de aplicar un comportamiento. La figura 10 especifica el diseño en detalle. En particular, brinda el detalle del comportamiento Campos Potenciales (ver[9] para más detalles), un conjunto de primitivas para definir comportamientos. Este subsistema está diseñado de forma extensible para incluir nuevas primitivas para definir comportamientos.

Se brinda la interfaz *IStrategy* para acceder desde otros subsitemas, de forma de independizar al resto del sistema de la implementación de los comportamientos.

Los componentes de este sistema son las siguientes:

- IStrategy: interfaz implementada por aquellos comportamientos que pueden ser utilizados, tal como el comportamiento Campos Potenciales, entre otros. Las funciones de esta clase son:

  - getSpeeds: esta función es utilizada para que a partir de las posiciones de las diferentes entidades del sistema se determine la velocidad a asignar a cada robot controlado, calculándolas a partir del comportamiento utilizado.

- AbstractStrategy: representa un comportamiento genérico. Los comportamientos específicos deberán extender la funcionalidad de esta clase.
  El atributo *referencedEntities* corresponde a los identificadores de las entidades que son utilizadas para calcular el resultado del comportamiento.
  El atributo *controledRobots* corresponde a los identificadores de los robost controlados por el comportamiento.

- PotentialFieldsStrategy: clase principal del comportamiento Campos Potenciales. En ella se almacena un conjunto de campos potenciales básicos (atributo fields) como pueden ser: campos potenciales generados en un semiplano, campos potenciales generados por punto o campos potenciales uniformes en todo el plano.

    - evaluateFields: esta función determina el valor de los campos presentes sobre un punto que es determina a partir de las coordenadas de éste que son recibidas como parámetro.

- Dot: puntos en el plano.

- OneField: clase que representa a la interfaz que es implementada por los diferentes campos potenciales. Posee asociada una colección ordenada de puntos del tipo *Dot*. Las funciones de esta clase son:

    - getField: esta función permite a partir de una posición determinar como influye el campo potencial en la posición provista. Retorna un vector que indica la influencia en la posición.

- FieldSemiPlane: representa un campo potencial generado por un segmento de recta. Este segmento genera un campo potencial sobre un semiplano de los dos que determina en el plano. Posee los atributos necesarios para que el campo quede determinado. El atributo *maxWidth* determina la distancia máxima sobre la que el campo tiene influencia en el plano a partir de la recta que genera el campo. El atributo *minWidth* determina la distancia mínima en la que el campo comienza a influir en el plano a partir de la recta genera el campo. El atributo *angle* determina el ángulo del campo potencial respecto al segmento de recta. El atributo *powStart* determina el módulo del campo a una distancia *minWidth* del segmento. El atributo *powEnd* determina el módulo del campo a una distancia igual a *maxWidth* del segmento. El semi-plano es el definido por los puntos (P_X) de la superficie para los cuales el seno del ángulo definido por $P_1P_2\hat{,}P_1P_x$es positivo. Los puntos P_1 y P_2 deben definirse en el orden correcto.

- FieldCircle: clase que es utilizada para representar el campo generado por un punto. Los atributos *minWidth*, *maxWidth*, *powStart* y *powEnd* son similares a los de la clase *FieldSemiPlane*, solamente con la salvedad que la distancias se calculan desde el punto generador del campo hasta el punto sobre el cual se calcula el campo.

- FieldUniform: clase que es utilizada para representar un campo uniforme en todo el plano. El atributo vX representa el valor del campo en el sentido del eje X del plano. El atributo vY representa el valor del campo en el sentida del eje Y del plano.

Además del comportamiento de tipo PotentialFieldsStrategy, se incluirán otros dos: de tipo Java y de tipo WaypointsStrategy.

El comportamiento de tipo Java (denominado CompiledJavaStrategy), refiere a la posibilidad de integrar cualquier comportamiento a partir de su definición en un archivo Java que será compilado e incluido en tiempo de ejecución en la aplicación como un comportamiento posible a utilizar. Para ver los detalles de dicho archivo, referirse a 7.2.1.

El comportamiento de tipo WaypointsStrategy refiere a aquel que determina que los robots controlados se desplacen a un conjunto ordenado de puntos en la superficie. Este comportamiento se implementará haciendo uso de campos potenciales, de forma similar a PotentialFieldsStrategy, con la salvedad que los campos potenciales influirán en el cálculo del comportamiento de forma temporal. Esto refiere a que cada punto a alcanzar será modelado mediante un campo potencial que se encontrará activo solamente cuando el robot debe alcanzar el punto que el campo modela. En el siguiente cuadro se presenta un pseudocódigo del cálculo de este comportamiento:

```
———————— Comienzo del pseudocódigo ————————
———————— Datos cargados: ————————
p0,p1,p2,...,pn-1 - Waypoints
epsilon - Distancia (en metros) cuando se considera que se alcanzó un waypoint
loop - Variable booleana que indica si se debe realizar un ciclo de recorrido por siempre
laps - Vueltas
speed - Velocidad (en metros por segundo)

———————— Variables utilizadas en el pseudocódigo: ————————
waypointIterator - Iterador de waypoints (se inicializa con el valor 0)
n - Cantidad de waypoints
p - Colección de puntos

———————— Pseudocodigo: ————————

ret={0,0,0}
//verifico si sigue corriendo
if (laps>0 || loop){
    if (distancia(posRobot, p[waypointIterator])
        //Llegue a un waypoint
        waypointIterator++
    if (waypointIterator == n )
    //termine una vuelta
        if (!loop)
            laps–
    //verifico devuelta que no haya terminado las vueltas
    if (laps>0 || loop){
        campo->setCenter(p[waypointIterator])
        ret = campo->getPower()
    }
}
return ret
———————— Fin del pseudocódigo ————————
```

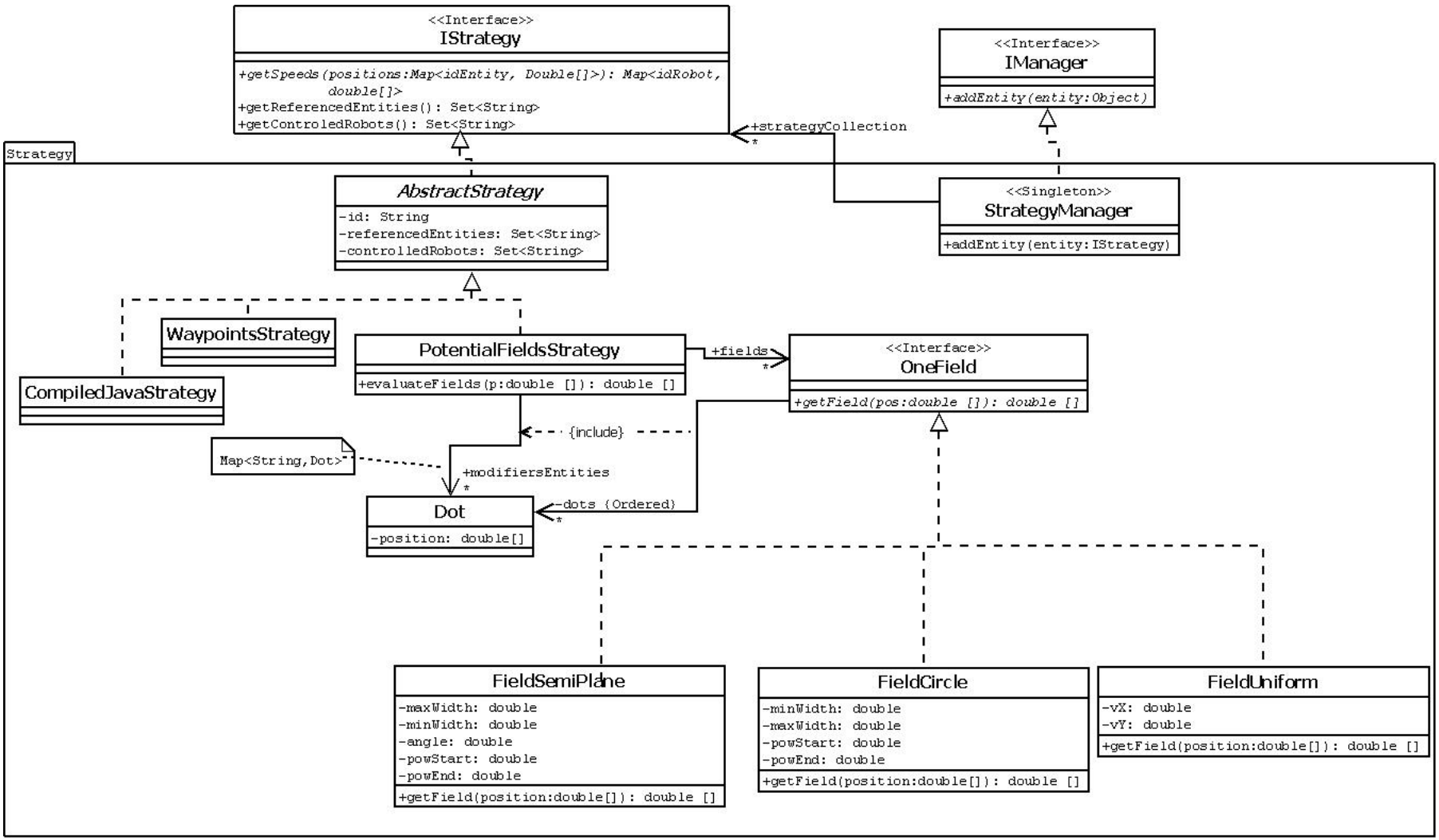

Figura 10: Diseño detallado del subsistema Strategy

Funcionamiento El funcionamiento general de este subsistema corresponde al cálculo de velocidades a asignar a cada entidad a patir del comportamiento que se encuentre activo en el sistema.

Función getSpeeds de la clase IStrategy A partir de la posición de las entidades del sistema, se deben retornar las nuevas velocidades que deben tomar los robots propios.

Para la implementación *PotentialFieldStrategy*, esta función deberá realizar principalmente dos acciones:

1. Actualizar posiciones: se actualiza la colección *modifiersEntities* a fin de actualizar la posición de las entidades que generan o influyen sobre los campos potenciales que se estén utilizando. Para ello, se itera en la colección *positions* recibida como parámetro y se obtienen las posiciones de las entidades necesarias.

2. Cálculo de campos: se calculan los campos en las posiciones de cada robot controlado por el comportamiento (colección *controledRobots*). Para realizar esto se invoca el método *evaluateField* con la posición del robot recibida dentro de la colección recibida como parámetro.

Función getReferencedEntities de la clase IStrategy Retorna el conjunto de identificadores de los robots referenciados por el comportamiento.

Función getControledRobots de la clase IStrategy Retorna el conjunto de identificadores de robots controlados por el comportamiento.

Función getField de OneField calcula el valor del campo potencial en las coordenadas provistas como parámetro. El cálculo interno varía de acuerdo al tipo de campo potencial.

4.3.7. Subsistema AppController

Este subsistema permite controlar la ejecución de los comportamientos cargados en el sistema. A partir de la información de un comportamiento, se obtienen los datos de las entidades tales como posiciones de los robots y obstáculos necesarios para el cálculo de dicho comportamiento. Este sistema es el encargado de modificar las velocidades de los robots lógicos que luego serán enviadas a los robots físicos. La figura 11 muestra el diseño del subsistema.

Los componentes de este sistema son las siguientes:

- AppController: componente principal del subsistema cuyo fin es activar y desactivar comportamientos. Este componente solicita los datos iniciales de la entidades que sean controladas por el comportamiento que se inicia. Se encuentra diseñada siguiendo el patrón Singleton [6] para asegurar que se cuente con una única instancia para controlar los comportamientos presentes en la aplicación. Posee un único atributo, *strategiesRunners*, que representa un mapa entre el identificador de cada comportamiento y la clase AppRunner que ejecuta dicho comportamiento. Las funciones de esta clase son:

  - activateStrategy: esta función es utilizada para activar un comportamiento a partir del identificador de éste.
  - deactivateStrategy: esta función es utilizada para desactivar un comportamiento a partir del identificador de éste.

- AppRunner: clase que ejecuta un comportamiento. Existen tantas instancias de esta clase como comportamientos activos en el sistema. Posee el atributo *strategy* que indica el comportamiento que se está ejecutando. Asimismo posee el atributo *dataSource* de tipo *IEntity* a fin de obtener los datos (posición) de las entidades que participan en el cálculo del comportamiento. Por último, el atributo *controls* indica el conjunto de los robots que son controlados por el comportamiento de forma de poder cambiar sus velocidades a partir del resultado del cálculo de dicho comportamiento. Las funciones de esta clase son:

  - start: inicia la ejecución de un comportamiento.
  - stop: detiene la ejecución del comportamiento actualmene en ejecución.

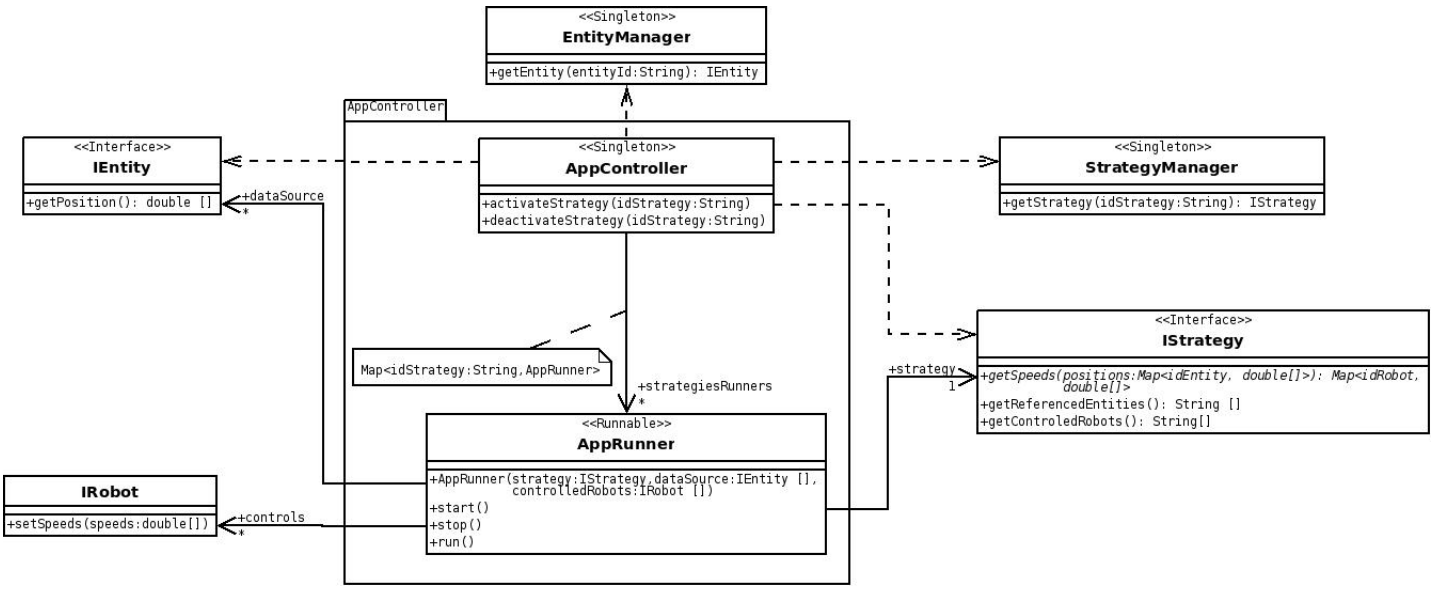

Figura 11: Diseño del subsistema *AppController*

Funcionamiento El funcionamiento general del subsistema corresponde a la ejecución de comportamientos. En él se maneja la correspondencia entre las entidades y las estrategias que utilizan sus posiciones así como el inicio y detención de dichos comportamientos.

Función activateStrategy de la clase AppController Esta función activa un comportamiento. Se pide la instancia de *IStrategy* al manejador de estrategias *StrategyManager*. A continuación, se le solicita a la instancia del comportamiento que retorne el conjunto de entidades referenciadas por el mismo. Para cada uno de los elementos del conjunto, se le solicita al manejador de entidades (*EntityManager*) la instancia para ser almacenada en un conjunto de entidades. Se crea una instancia de AppRunner con un hilo de ejecución asociado. Finalmente, se inicia este hilo quedando en ejecución el comportamiento deseado.

Función deactivateStretegy de la clase AppController El funcionamiento de esta función corresponde a la desactivación de un comportamiento activo. En ella únicamente se invoca a la función stop de la instancia *AppRunner* que posee al identificador recibido como parámetro y finalmente se elimina la instancia de *AppRunner*.

Función run de la clase AppRunner En esta función se mantiene la ejecución de un comportamiento cada cierto período de tiempo. La ejecución básica es:

- Obtener las posiciones de las entidades.
- Invocar a la función getSpeeds de la estrategia asociada .
- Por último, se envían las nuevas velocidades a los robots físicos.

Función stop de la clase AppRunner Esta función detiene el cálculo del comportamiento.

4.4. Colaboraciones entre subsistemas

A continuación se explica el flujo de datos en el sistema. Existen varios tipos de flujos de datos:

- *Inicialización*: Al comenzar la ejecución de la aplicación se solicita al usuario que introduzca información necesaria para cargar e inicializar el sistema robótico.
- Activacion: La activación de un comportamiento.
- Ejecución: Cuando el sistema está inicializado y algun comportamiento se encuentra activo, este flujo de datos permite controlar los robots del sistema.

A continuación se describe y diagrama cada flujo de datos.

4.4.1. Carga de un sistema robótico

En la figura 12 se observa el diagrama de flujo de datos cuando el usuario inicia y activa el sistema. Las etapas del flujo de datos son:

1. En la primer etapa, el usuario indica mediante el subsistema *GUI,* que desea cargar un sistema robótico. Debe indicar la ubicación del archivo de definición del sistema.

2. En esta etapa el subsistema *Controller* invoca al subsistema *Loader* con la ruta del archivo de definición a fin de que sea cargado en el sistema.

3. Corresponde a la carga de las instancias del subsistema *Entity*. Asimismo se crean las instancias que modelan a la comunicación con los *Sistemas de Actuación*.

4. Se crean instancias, en estado Desactivado, para todos los comportamientos definidos en el sistema robótico.

5. Se crean los sensores lógicos. Las comunicaciones de éstos con los *Sistemas de Sensado* aún no son activadas.

6. Se inicializan las entidades y robots del sistema.

7. Se crea la relación entre las entidades y robots con los componentes del subsistema *Position* que los controlan. Esto asegura que el subsistema *Entity* recibirá actualizaciones provenientes del subsistema *Position* cuando se comience el sensado del sistema.

8. Se inicializan los sensores lógicos. Para aquellos sensores que requieren una comunicación con un *Sistema de Sensado*, se crean los canales de conexión. Para cada uno de los componentes del subsistema se crea un hilo de ejecución propio.

9. Para los componentes que requieren datos del subsistema *Entity*, se crean las comunicaciones con las entidades o robots, por ejemplo para los sensores predictores.

10. Se crea la comunicación entre robots lógicos y los *Sistemas de Actuación*.

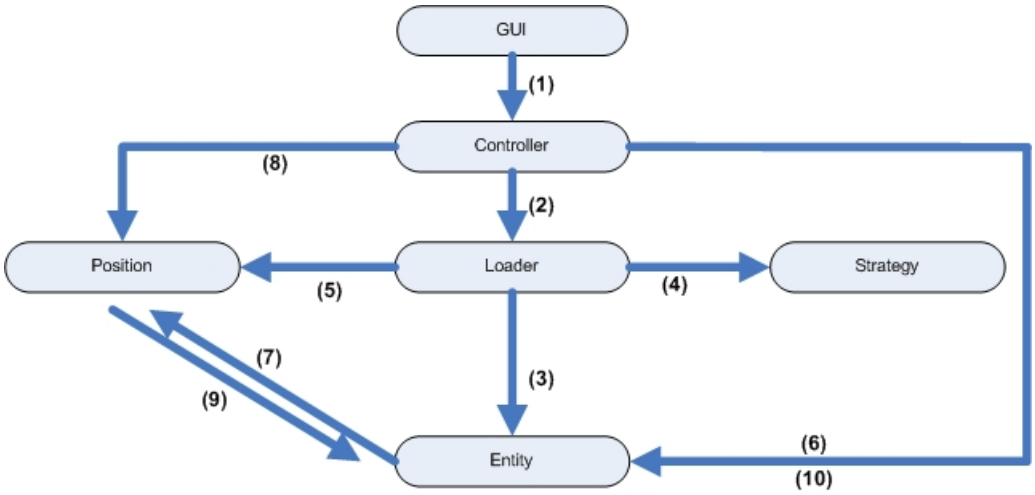

Figura 12: Flujo de datos en la inicialización de un sistema robótico.

Como resultado de la ejecución de este flujo, la aplicación comienza a recibir datos de los *Sistemas de Sensado*. De igual forma, los robots pueden comenzar a ser controlados por el sistema.

4.4.2. Activación de un comportamiento

Cuando el usuario desea activar un comportamiento, se produce el flujo de datos que se observa en la figura 13. Las etapas de éste son:

1. El usuario indica mediante el subsistema *GUI* que desea activar un comportamiento. Debe indicar el identificador del comportamiento a activar.
2. El subsistema *Controller* invoca al subsistema *AppController* para que este último active el comportamiento solicitado.
3. El subsistema *AppController* obtiene el comportamiento a activar invocando al subsistema *Strategy*.
4. Esta etapa contiene la invocación al subsistema *Entity* desde el subsistema *AppController* para obtener las instancias que modelan las entidades y robots a ser utilizadas en la evaluación del comportamiento, a fin de solicitarles a éstas datos tales como sus posiciones, velocidad entre otros.
5. Si bien no se encuentra visible en el diagrama, esta etapa crea el hilo de ejecución del comportamiento activado.

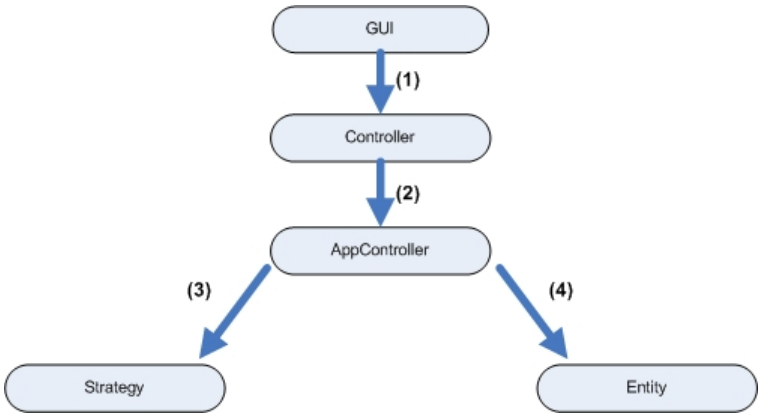

Figura 13: Flujo de datos de la activación de un comportamiento.

4.4.3. Evaluación de comportamientos

Una vez activados los comportamientos que se desean, comienza la ejecución de todos los componentes del sistema de forma cohesiva. En este caso, se generan tres flujos de datos que se pueden observar en la figura 14. Existe un flujo de datos para la actualización de posiciones (1 y 2); otro, que comprende la evaluación, ejecución del comportamiento que aplique y la actualización de los sensores lógicos que requieran velocidades (etapas 3, 4, 5, 6 y 7) .

A continuación se explica el flujo de actualización de posiciones:

1. Los componentes del subsistema *Position* reciben datos sobre las posiciones de las entidades que son controladas en el mundo físico.
2. Se envían las posiciones de las entidades recibidas, al subsistema *Entity*.
3. El subsistema *AppController* solicita las posiciones a las entidades y robots del subsistema *Entity* que serán utilizadas por el comportamiento que se ejecutará. En la figura, se identifica con (A).
4. Se evalúa el comportamiento utilizando las posiciones obtenidas en la etapa anterior.
5. El subsistema *AppController* actualiza las velocidades de los robots, del subsistema *Entity*, que son controlados por el comportamiento ejecutado.
6. El subsistema *Entity* que provee datos a los componentes del subsistema *Position*, tales como sensores de prediccion, notifica las velocidades actualizadas por el comportamiento evaluado.
7. El subsistema *Entity* envía las velocidades a los robots físicos actualizados.

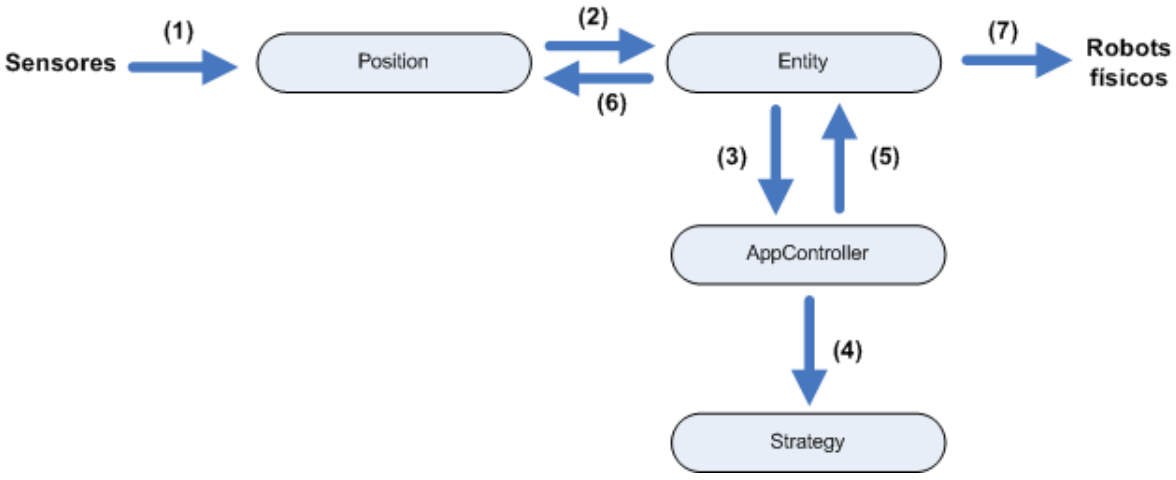

Figura 14: Flujo de datos durante la ejecución de la aplicación.

5. Vista de Procesos

5.1. Procesos del Sistema

La arquitectura de procesos del sistema, tal como lo muestra la figura 15, está compuesta únicamente por el proceso del sistema principal (Sistema EasyRobots). A su vez de éste participan diversos hilos para cada función principal del sistema.

Se cuenta con el hilo AppRunner, encargado de la ejecución de los comportamientos activos.

Así mismo se cuenta con el conjunto de hilos *Position*, destinados a la recepción de datos de los sensores físicos. La ejecución de este hilo está definida en la implementación específica del Sistema de Sensado utilizado.

Opcionalmente, se encuentran los hilos PhysRobot destinados a la comunicación con los robots físicos. Estos hilos son opcionales dependiendo de la implementación particular del Sistema de Actuación utilizado.

Todos los hilos descritos anteriormente comparten la memoria del sistema, en el que se almacenan las entidades, comportamientos y otros componentes de la aplicación robótica, las cuales son accedidas concurrentemente.

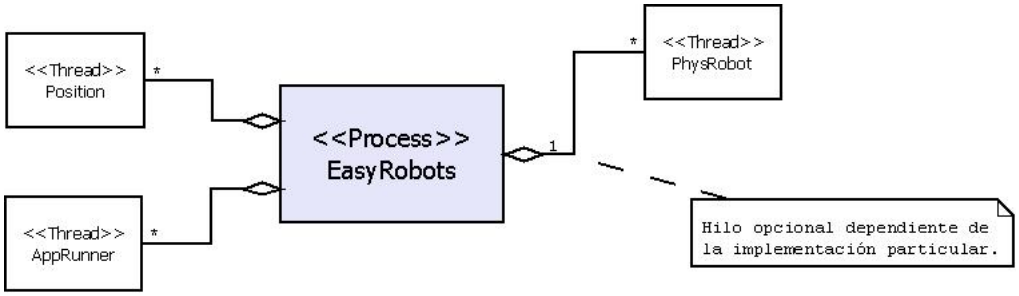

Figura 15: Diseño de procesos del sistema

5.2. Procesos de la Aplicación

En esta sección se explicará el proceso de inicialización de la aplicación.
El proceso se subdivide en los siguientes pasos:

1. El controlador presente en el subsistema *Controller* invoca al cargador del subsistema *Loader* para que éste realice la carga de los datos, proceso llamado *Carga*. En este proceso además, luego de cargar las entidades lógicas y las instancias que representan las entidades físicas si estás se encuentran presentes, se crean las relaciones entre éstas. El orden de la carga de las componentes de la aplicación robótica es:

   a) Se cargan los controladores de los robots físicos.

   b) Se cargan los comportamientos indicados que comienzan inactivos.

   c) Se cargan las entidades que representan a los robots de la aplicación y otras componentes tales como pelotas y obstáculos.

   d) Se cargan las instancias que representan los sensores.

2. Luego de poseer todas las instancias que componen la aplicación robótica cargadas, se crean las relaciones que vinculan a las instancias. A este proceso se le llama *Inicialización*. Este proceso se subdivide en diferentes sub-procesos de acuerdo al tipo de la instancia:

   a) Cada entidad se agrega como observadora del sensor que le provee datos sobre su posición.

   b) Cada instancia que represente un sensor predictor se agrega como observadora de la entidad de la cual obtiene la velocidad y posición.

   c) Las instancias de sensores globales crean los canales de comunicación con los sensores en el medio físico.

   d) Las instancias de las fusiones de sensores crean las referencias a los sensores que fusionan.

6. Vista de Distribución

6.1. Introducción

Está sección describe una o más configuraciones físicas sobre las cuales se realiza la distribución del software y es ejecutado, así como la infraestructura necesaria para su instalación. Se describe el escenario general del sistema y se muestran además un diagrama de distribución de una aplicación específica de forma de clarificar la distribución del sistema EasyRobots.

6.2. Distribución

La figura 16 presenta el escenario de distribución esperado para la instalación del framework. El mismo se ubica en el contexto de una organización, sobre una LAN.

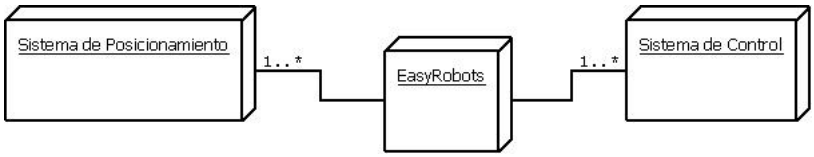

Figura 16: Escenario de distribución general

A continuación se describen los nodos presentes en la figura:

EasyRobots Este es el sistema a desarrollar.

Subsistema de Posicionamiento Este nodo corresponde a los Sistemas de Sensado detallados en el modelo de una aplicación específica del framework. El sistema EasyRobots deberá comunicarse con estos para obtener datos de posicionamiento de las entidades definidas.

Sistema de Control Corresponde a los Sistemas de Comunicación con los robots físicos definidos en una aplicación específica del framework. El sistema EasyRobots se comunicará con éstos para enviar nuevas directivas a los robots físicos.

El diagrama presentado anteriormente corresponde a una distribución genérica de una aplicación utilizando el framework. A continuación se presenta un posible diagrama de distribución específico de una aplicación desarrollada utilizando EasyRobots. Se puede ver un PC corriendo el sistema EasyRobots, la cual se comunica con otros 2 PCs; uno de ellos tiene acceso a una cámara de video para la obtención de datos globales de las entidades, y la otra tiene acceso a la comunicación con los robots de la aplicación. En este caso la aplicación desarrollada tiene solo una fuente de posicionamiento y un único método de comunicación con robots, sin embargo el sistema esta diseñado de forma de permitir más conexiones simultáneas con estos tipos de sistemas.

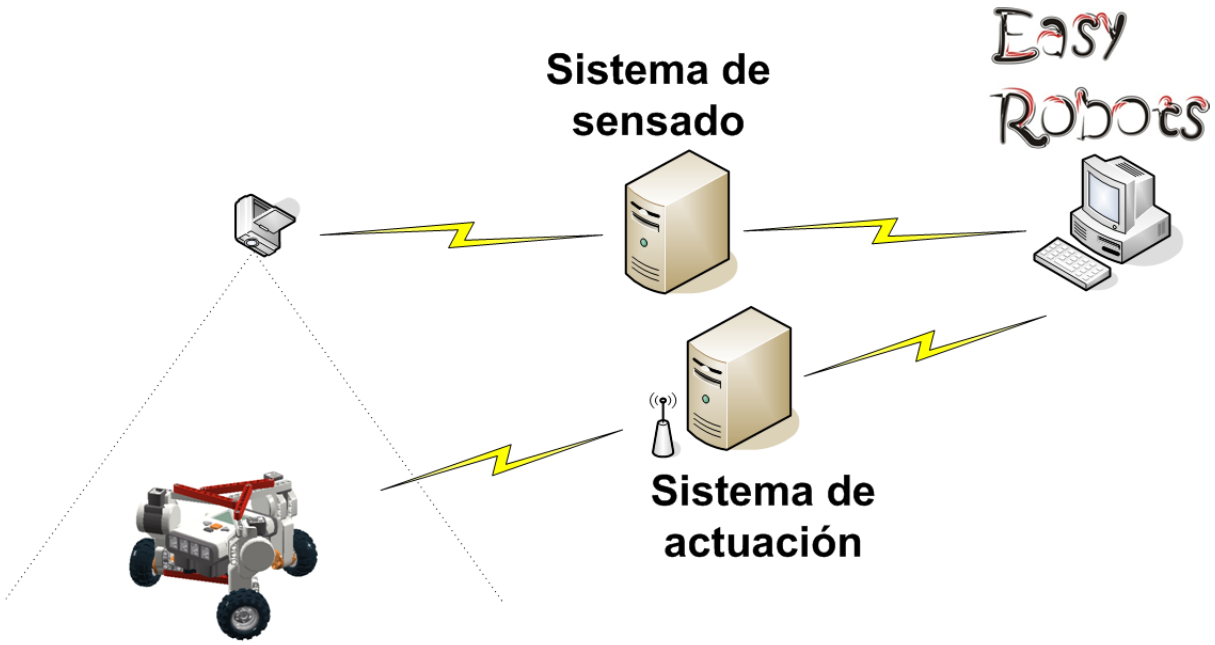

Figura 17: Escenario de distribución general

7. Vista de Datos

Dadas las características del sistema a construir, no se establece como requerimiento la persistencia de datos. Sin embargo, al necesitarse cargar una aplicación robótica se necesita una fuente de datos para este fin. Estos datos deben indicar las características que definen completamente las entidades y relaciones de la aplicación. Se utilizará un archivo con formato XML para almacenar una aplicación robótica de usuario.

Este archivo estará compuesto por tres grandes partes:

Entidades Corresponden a robots y otros elementos físicos como pelotas y obstáculos. Para los robots propios se deberá brindar información de comunicación con el robot físico.

Comportamientos Se definirán los *comportamientos* en alto nivel utilizando las primitivas brindadas por el sistema (campos potenciales, diagramas de Voronoi, etc).

Posicionamiento Se definirán los Sistemas de Sensado a utilizar, detallando los atributos necesarios para lograr una comunicación exitosa con estos sistemas.

A continuación se detallará la estructura del archivo XML. Se explicará desde el mayor nivel de abstracción del archivo hasta el detalle de cada componente.

7.1. Información de las Entidades

Para definir las entidades del sistema, se pueden utilizar diferentes tipos de acuerdo al tipo de entidad.

7.1.1. Tipo Entity

El tipo más general que se puede utilizar es *Entity*. Para este tipo de entidad se deben definir los siguientes parámetros:

- Identificador de la entidad.
- Identificador del sensor del cual obtiene cambios en su posición.
- Tipo de la entidad que se define.

El siguiente es un ejemplo de definición:

```
<entity xsi:type="Entity" id="entityId" positionId="sensorId" />
```

7.1.2. Tipo GeneralEntity

Otro tipo de entidad posible es *GeneralEntity*. Si bien este tipo de entidad es más específica que el tipo *Entity*, su definición no requiere información adicional.

7.1.3. Tipo AbstractRobot

También se cuenta con el tipo de entidad *AbstractRobot*, utilizado para especificar que la entidad que se define es un robot.

De utilizarse este tipo de entidad, se deben definir los siguientes parámetros:

- Velocidad máxima de cada rueda (en metros por segundo). Debe ser considerado que no se permiten implementaciones con velocidades máximas de ruedas diferentes para cada una. De no especificarse este valor, se considerará que no se posee un límite máximo de velocidad.
- Velocidad máxima de desplazamiento del robot. Se especifican tres valores ordenados: el primero especifica la velocidad máxima en el eje X del sistema (en metros por segundo), el segundo especifica la velocidad máxima en el eje Y (en metros por segundo) y el tercero indica la velocidad máxima de rotación sobre el eje vertical por su centro (en radianes por segundo). De no especificar estos valores, se considerará que no se poseen velocidades máximas.

- Implementación de robot físico asociado. Este elemento posee tipos internos a utilizar. Su utilización se explicará a continuación. Este atributo es opcional a fin de realizar casos de prueba sin implantaciones físicas de robots.

Como se explicó en el párrafo anterior, el robot físico asociado, en caso de definirse, requiere un tipo específico. Un ejemplo de los tipos de robots físicos a tratar, que será incluido en la aplicación desarrollada, es el *NXTRobot*. Para este tipo de robot, se deben definir los siguientes parámetros:

- Identificador del controlador que enviará los mensajes pertinentes al robot físico.
- Dirección física del robot (en este caso, la dirección MAC[1] del robot).

El siguiente es un ejemplo de definición de un robot lógico con un robot físico asociado:

```
<entity
    xsi:type="AbstractRobot"
    id="idRobot1"
    positionId="positionId1"
    maxwheelrotation="1.3" >
    <physicalRobot
        xsi:type="NXTRobot"
        physicalAddress="00:16:53:01:4E:3B"
        physicalControllerId="physControllerId1"
    />
    <maxspeed>0.5</maxspeed>
    <maxspeed>0.7</maxspeed>
    <maxspeed>1.5</maxspeed>
</entity>
```

Para que las velocidades en el sistema de coordenadas utilizado sean convertidas a velocidades de ruedas, se debe definir un tipo de robot particular. Para ello, a través del tipo de entidad se debe indicar el tipo de robot particular que se modelará.

OmnidirectionalRobot Un ejemplo de robot posible a utilizar es el *OmnidirectionalRobot*. Para este tipo de robot, además se deben especificar los siguientes parámetros:

- Se deben describir las ruedas del robot, éste debera contar con al menos 3 ruedas para poder ser omnidireccional y sea validado por el programa. Las características de las ruedas a utilizar se detallan a continuación y se pueden ver en la imagen 18 :
  - El ángulo del eje de los rodillos pasivos respecto del eje de la rueda (atributo *alpha*).
  - El ángulo entre el eje 0X del sistema de coordenadas y la rueda (atributo *beta*).
  - El ángulo entre el eje 0X y el eje de la rueda (considerando al eje como el vector entre el centro de masa del robot y el centro de la rueda), en el atributo *gamma*.
  - La distancia del centro de masa del robot al centro de la rueda (atributo d). Esta distancia está especificada en la unidad metros.
  - El radio de la rueda utilizando el atributo *radius*.

Todos los ángulos que se definan, deben estar en unidad Radianes.

[1] Media Access Control: Identificador único asignado a cada adaptador de red o tarjeta de interfaz de red.

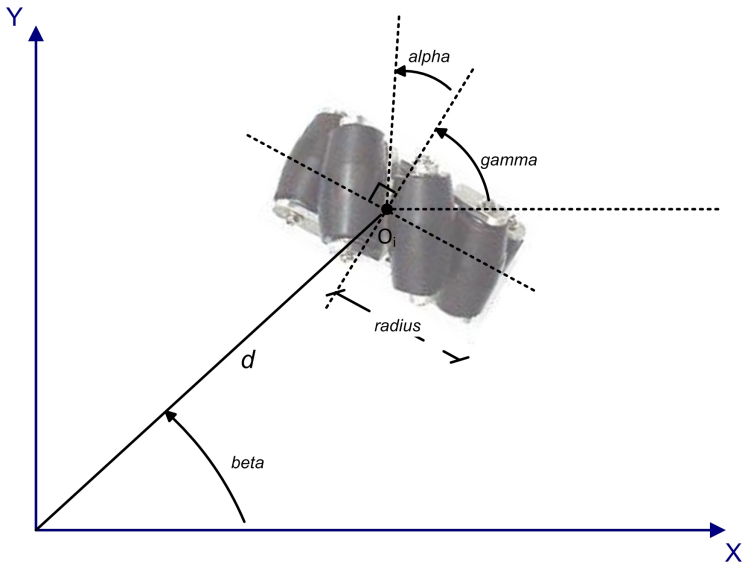

Figura 18: Características de las ruedas omnidireccionales

El siguiente es un ejemplo que contempla la definición completa de un robot omnidireccional de tres ruedas:

```
<entity
    xsi:type="AbstractRobot"
    id="idRobot1"
    positionId="positionId1"
    maxwheelrotation="1.3" >
    <physicalRobot
        xsi:type="NXTRobot"
        physicalAddress="00:16:53:01:4E:3B"
        physicalControllerId="physControllerId1"
    />
    <maxspeed>0.5</maxspeed>
    <maxspeed>0.7</maxspeed>
    <maxspeed>1.5</maxspeed>
    <wheel
        alpha="1.5707"
        beta="0.0"
        d="0.05"
        gamma="0.0"
        radious="0.032"
    />
    <wheel
        alpha="1.5707"
        beta="2.0943951"
        d="0.05"
        gamma="2.0943951"
        radious="0.032"
    />
    <wheel
        alpha="1.5707"
        beta="4.1887902"
        d="0.05"
        gamma="4.1887902"
        radious="0.032"
    />
</entity>
```

BiWheeledRobot El robot *BiWheeledRobot* modela un robot de dos ruedas. Para su definición se requieren los siguientes parámetros:

- Radio (medido en metros) del robot, indicando la distancia de cada rueda al centro del robot
- Radio (medido en metros) cada rueda del robot.

El siguiente es un ejemplo que contempla la definición completa de un robot de dos ruedas:

```
<entity
    id="idRobot"
    xsi:type="BiWheeledRobot"
    maxwheelrotation="0.0"
    positionId="sensor1"
    robotRadius="0.1"
    robotWheelRadius="0.032" >
</entity>
```

7.2. Información de los Comportamientos

De forma similar a la definición de entidades, los comportamientos se definen utilizando diferentes tipos.

7.2.1. Tipo Strategy

El objeto mas general para definir un comportamientos requiere de los siguientes parámetros:

- Identificador del comportamientos. Este parámetro es utilizado por el manejador de comportamientos para acceder a la comportamiento específico.
- Conjunto de identificadores de robots que son controlados por el comportamientos.

De esta forma, un comportamientos general se representa en el archivo formato XML de la siguiente forma:

```
<strategy
    id="EstrategyId"
    xsi:type="Strategy">
    <controlledRobot>idRobot1</controlledRobot>
    <controlledRobot>idRobot2</controlledRobot>
</strategy>
```

Tipo PotentialFieldsStrategy Un tipo de comportamiento posible a utilizar, que será incluído en la versión final de la aplicación, es el comportamiento de Campos Potenciales (*PotentialFieldsStrategy*, como se define en la aplicación). Para este tipo, además de los parámetros en común, se deben definir los siguientes parámetros:

- Campo potencial que participa en el cálculo de comportamiento (atributo *field*). Los diferentes tipos de campos poseen un único parámetro en común:
  - Conjunto ordenado de puntos que son utilizados para el cálculo del campo. Los puntos pueden ser definidos de dos formas distintas: a partir de un identificador de una entidad de la cual se obtiene su posición en la superficie al momento de realizar el cálculo o a partir de un par de parámetros numéricos que indican un punto en la superficie en coordenadas del sistema.

  Los campos adicionalmente deben ser definidos de un tipo específico, a saber:

  Tipo FieldCircle Define un campo potencial Circular, generado a partir de un punto en la superficie. Este tipo de campo requiere los siguientes parámetros:

  - La distancia máxima a partir del punto indicando el alcance del campo (atributo *maxwidth*).
  - La distancia mínima a partir del punto en la que el campo comienza a influir en el cálculo (atributo *minwidth*).
  - Valor del campo en la distancia mínima (atributo *powstart*).
  - Valor del campo en la distancia máxima (atributo *powend*).

    El valor del campo en un punto que se encuentre a una distancia del punto entre la mínima y la máxima definida se obtiene de interpolar el valor del campo en la distancia mínima (*powStart*) y la distancia máxima (*powEnd*). Si se requiere que el campo sea atractor, ambos valores deben ser negativos y viceversa. El punto generador del campo es determinado a partir del primer punto en la colección ordenada de puntos del comportamiento. En la figura 19 se representa lo anterior.

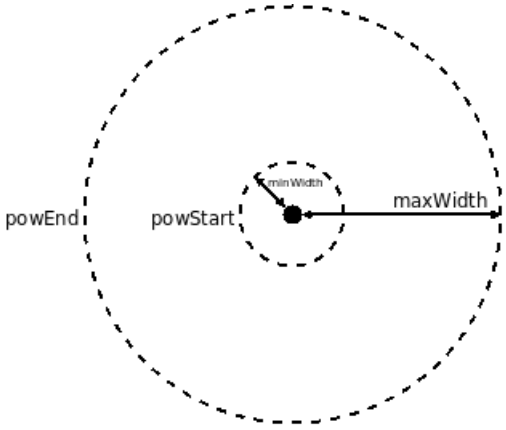

Figura 19: Atributos que definen el campo potencial circular.

Este tipo de campo potencial es representado en el archivo XML de la siguiente forma:

```
<field
    xsi:type="FieldCircle"
    minWidth="0.1"
    maxWidth="0.5"
    powEnd="-0.1"
    powStart="-0.3">
    <dot>
        <id>idBall</id>
    </dot>
</field>
```

Tipo FieldUniform Representa un campo uniforme en el plano. Este tipo requiere de dos parámetros para definirse completamente:

- Valor del campo en la coordenada X del sistema (atributo vX).
- Valor del campo en la coordenada Y del sistema (atributo vY).

Este tipo de campo potencial se representa en el archivo XML de la siguiente forma:

```
<field
    xsi:type="FieldUniform"
    vX="0.0"
    vY="0.0">
</field>
```

Tipo FieldSemiPlane Representa un campo generado por una semi-recta sobre uno de los semi-planos que ésta genera. Este tipo de campo requiere de los siguientes parámetros para definirse completamente:

- Ángulo del campo en la superficie respecto de la semi-recta generadora del campo (atributo *angle*)
- La distancia máxima a partir de la semi-recta indicando el alcance del campo (atributo *maxwidth*).
- La distancia mínima a partir de la semi-recta en la que el campo comienza a influir en el cálculo (atributo *minwidth*).
- Valor del campo en la distancia mínima (atributo *powstart*).

- Valor del campo en la distancia máxima (atributo *powend*).

  La semi-recta, es determinada por los dos primeros puntos en la colección ordenada de puntos asociada al campo. El semi-plano es el definido por los puntos (P_X) de la superficie para los cuales el seno del ángulo definido por $P_1P_2\hat{,}P_1P_x$es positivo. De forma similar que para el campo de tipo FieldCircle, el valor del campo en un punto a una distancia de la semi-recta que se encuentra entre los valores *maxwidth* y *minwidth* se obtiene a partir de la interpolación del valor del campo en dichas distancias. Nuevamente, si se desea que el campo sea atractor hacia la semi-recta, los valores de *powStart* y *powEnd* deben ser negativos y viceversa. La figura representa lo anterior.

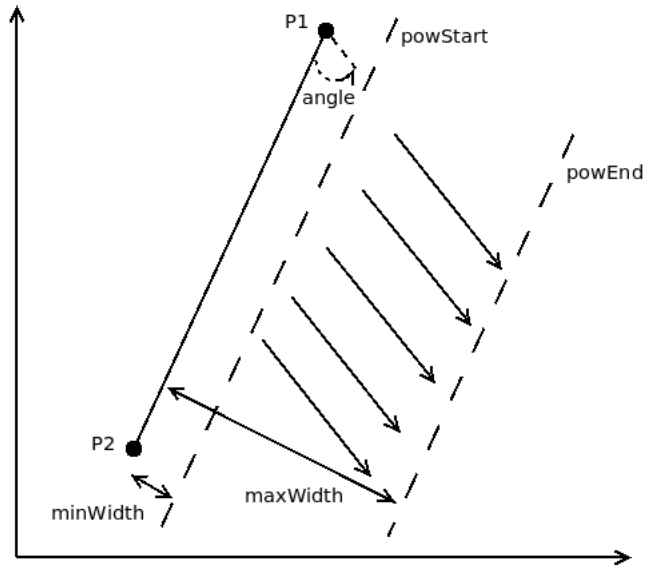

Figura 20: Atributos que definen el campo potencial generado por una semi-recta.

Este tipo de campo potencial se representa en el archivo XML de la siguiente forma:

```
<field
    xsi:type="FieldSemiPlane"
    angle="3.14159"
    minWidth="0.01"
    maxWidth="0.2"
    powEnd="0.3"
    powStart="0.4">
    <dot>
        <position>2</position>
        <position>15</position>
    </dot>
    <dot>
        <id>idRobot1</id>
    </dot>
</field>
```

El siguiente es un ejemplo de un comportamiento de campos potenciales completamente definido:

```
<strategy
    xsi:type="PotentialFieldsStrategy"
    id="strategyId">
    <controlledRobot>idRobot</controlledRobot>
    <field
        xsi:type="FieldCircle"
        minWidth="0.1"
        maxWidth="5"
        powStart="-3"
        powEnd="-3">
        <dot>
            <id>idBall</id>
        </dot>
    </field>
    <field
        xsi:type="FieldSemiPlane"
        angle="3.14159"
        minWidth="0.01"
        maxWidth="0.2"
        powEnd="0.3"
        powStart="0.4">
        <dot>
            <position>2</position>
            <position>15</position>
        </dot>
        <dot>
            <id>idRobot1</id>
        </dot>
     </field>
</strategy>
```

Tipo JavaStrategy Para definir este tipo de comportamiento se debe incluir un archivo Java. Para ello el usuario de la aplicación debe definir una clase Java con una estructura particular para luego detallar la ruta a la clase creada a fin de ser compilada en tiempo de ejecución por el sistema.

La estructura de la clase a definir debe ser la siguiente:

```
public class Example extends com.easyRobots.engine.strategy.AbstractStrategy{
    //Código adicional creado por el usuario

    @Override
    public java.util.Map<String, double[]> getSpeeds(java.util.Map<String, dou-
ble[]> positions, java.util.Map<String, double[]> speeds) {
        // Código del comportamiento
        return resultado_del_comportamiento;
    }
}
```

Como se puede observar, el usuario debe impelementar la función *getSpeeds* a fin de retornar las velocidades a asignar a cada robot a partir de las posiciones de las entidades en la superficie.

Adicionalmente a especificar la ruta de la clase, el usuario deberá brindar la lista de identificadores de entidades que son requeridas para el cálculo del comportamiento. Esta lista, es utilizada por la función *getSpeeds* a través del parámetro *positions*.

Para indicar que se desea integrar una clase Java a la aplicación, el usuario debe indiciarlo a través del archivo XML que se utilice. Ésto se representa en el archivo XML de la siguiente forma (observar el atributo *filePath*):

```
<strategy
    xsi:type="JavaStrategy"
    id="javaTestStrategyId"
    filePath="./Clase.java">
    <controlledRobot>idRobot1</controlledRobot>
    ...
    <controlledRobot>idRobotN</controlledRobot>
    <referencedEntity>idReferencedEntity1</referencedEntity>
    ...
    <referencedEntity>idReferencedEntityN</referencedEntity>
</strategy>
```

Tipo WaypointsStrategy Para definir un comportamiento de este tipo, se deben especificar los siguientes parámetros:

- Margen de error. Refiere al error máximo permitod entre la posición del robot y el punto a alcanzar. Este parámetro es representado mediante el atributo *epsilon*.
- Cantidad de vueltas. Refiere a la cantidad de veces que se debe repetir el recorrido por cada uno de los puntos. Este parámetro se encuentra representado en el atributo *laps*.
- Ciclico. Refiere a si el recorrido es cíclico. Esto es, si el robot no se debe permanecer realizando el circuito de puntos de forma indefinida. Se encuentra representado en el atributo *loop*.
- Velocidad. Refiere a la velocidad del robot para desplazarse de un punto a otro. Se expresa en metros por segundo mediante el uso del atributo *speed*.
- Secuencia de puntos. Refiere a la secuencia de puntos, de forma ordenada, que el robot debe alcanzar. Cada punto puede ser estático, o dependiente de la posición de otra entidad.

Este tipo de comportamiento se define mediante el formato de XML de ejemplo expuesto en el siguiente cuadro:

```
<strategy
    id="s1"
    xsi:type="WaypointsStrategy"
    epsilon="0.1"
    laps="1"
    loop="false"
    speed="1">
    <controlledRobot>idRobot</controlledRobot>
    <waypoint>
        <position>5</position>
        <position>0</position>
    </waypoint>
    <waypoint>
        <id>entityId</id>
    </waypoint>
    <waypoint>
        <position>0.0</position>
        <position>0.0</position>
     </waypoint>
</strategy>
```

7.3. Información de los Sensores

Los sensores, se representan lógicamente mediante la clase *AbstractSensor*. Estos sensores son definidos mediante tipos mas específicos como se explicará a continuación.

7.3.1. Tipo AbstractSensor

El objeto mas general dentro del sistema de sensores es de tipo *AbstractSensor*. Requiere los siguientes parámetros:

- Identificador del sensor (atributo *id*). Este parámetro es utilizado por el manejador de sensores para acceder a éste.
- Confianza del sensor. Este parámetro indica la confianza (medida entre los valores 0 y 1)sobre las mediciones arrojadas por el sensor. Es utilizado en la fusión de sensores.
- Período del sensado. Este parámetro indica el intervalo de tiempo (medido en milisegundos) entre mediciones del sensor.

Este sensor se representa en XML de la siguiente forma:

```
<sensor
    confidence="0.4"
    id="sensorId"
    period="1"
/>
```

7.3.2. Tipo GlobalPositionSensor

Este tipo de sensor extiende el tipo *AbstractSensor* indicando que el sensor definido realiza mediciones de posiciones globales respecto del sistema. No requiere de parámetros adicionales respecto de los definidos en *AbstractSensor*.

Este tipo de sensor de define en XML de la siguiente forma:

```
<sensor
    xsi:type="GlobalPositionSensor"
    confidence="0.7"
    id="sensorId"
    period="0.03"
/>
```

Un tipo de sensor global definido en la aplicación, en su versión final, es el *DoraemonClient*. Dicha instancia, utilizada para representar el sistema de visión Doraemon[2], es representada en XML de la siguiente forma:

7.3.3. Tipo SensorFusion

Este tipo de sensor, al igual que *GlobalPositionSensor*, extiende el tipo *AbstractSensor* a fin de especificar una fusión de sensores. Los parámetros para definir estos sensores son:

- El identificador de la entidad que controla el sensor (atributo *managedEntityId*).
- Colección de identificadores de los sensores que son fusionados (atributo *fusionedSensorId*).

Este tipo de sensor se define en XML de la siguiente forma:

[2]Sistema de visión desarrollado por Jacky Baltes bajo licencia pública GNU. Más información en http://sourceforge.net/projects/robocup-video.

```
<sensor
    xsi:type="SensorFusion"
    confidence="0.9"
    id="idSensor"
    managedEntityId="entityId"
    period="0.5">
    <fusionedSensorId>sensor1Id</fusionedSensorId>
    <fusionedSensorId>sensor2Id</fusionedSensorId>
    <fusionedSensorId>sensorNId</fusionedSensorId>
</sensor>
```

Un tipo de fusión de sensores posible a utilizar, que será incluída en la versión final de la aplicación, es el fusionado utilizando el promedio ponderado de las mediciones de sensores. Dicho tipo de fusión es implementado en la aplicación mediante la clase *AverageSensorFusion*. Esta fusión no requiere de parámetros adicionales a los de *SensorFusion*. A continuación se especifica la definición de este tipo de fusión en un archivo XML:

```
<sensor
    xsi:type="AverageSensorFusion"
    confidence="0.8"
    id="idSensor"
    managedEntityId="entityId"
    period="0.6">
    <fusionedSensorId>sensor1Id</fusionedSensorId>
    <fusionedSensorId>sensor2Id</fusionedSensorId>
    <fusionedSensorId>sensorNId</fusionedSensorId>
</sensor>
```

El tipo de fusión de sensores restante implementado es el *FashionSensorFusion*. Este consiste en seleccionar el dato sensado por el sensor más prioritario y disponible de los fusionados. Para definir este tipo de fusión se requiere adicionalmente a los parámetros de *SensorFusion* el siguiente:

- Ventana de tiempo utilizada para determinar la validez de los datos sensados por los sensores fusionados.

El código XML que define este tipo de fusión es el siguiente:

```
<sensor
    xsi:type="FashionSensorFusion"
    confidence="0.8"
    id="idSensor"
    managedEntityId="entityId"
    checkThreshold="50"
    period="0.6">
    <fusionedSensorId>sensor1Id</fusionedSensorId>
    <fusionedSensorId>sensor2Id</fusionedSensorId>
    <fusionedSensorId>sensorNId</fusionedSensorId>
</sensor>
```

Considerar que el orden de definición de los sensores considerados, utilizando el elemento *fusionedSensorId*, es el orden de consulta durante la fusión de los datos.

7.3.4. Tipo PredictionSensor

Este tipo de sensor extiende al tipo AbstractSensor. Representa un predictor de la futura posición de una entidad a partir de la velocidad que ésta posee al momento de realizar el cálculo y la última posición conocida. Los parámetros para definir un predictor son:

- Identificador de la entidad a la que se le predice su futura posición.
- Identificador del sensor que utiliza como referencia para obtener la posición de la entidad. Es un parámetro opcional.

El siguiente es un ejemplo de un sensor predictor definido en XML:

```
<sensor
    xsi:type="PredictionSensor"
    managedEntityId="idRobot"
    referenceSensorId="sensorId"
    id="sensorPredictor"
    confidence="1"
    period="10">
</sensor>
```

Referencias

[1] IBM Rational Software. Rational unified process. Retrieved 19 May 2009 http://www-306.ibm.com/software/awdtools/rup/. URL `http://www-306.ibm.com/software/awdtools/rup/`.

[2] IBM Rational Software Corporation. Rational unified process. Retrieved 19 May 2009 http://www.ts.mah.se/RUP/RationalUnifiedProcess/index.htm. URL `http://www.ts.mah.se/RUP/RationalUnifiedProcess/index.htm`.

[3] Rafael Sisto and Santiago Martínez. Modelo de casos de uso. Technical report, Instituto de Computación, Facultad de Ingeniería, Universidad de la República, Uruguay, 2009.

[4] Rafael Sisto and Santiago Martínez. Especificación de requerimientos de software. Technical report, Instituto de Computación, Facultad de Ingeniería, Universidad de la República, Uruguay, 2009.

[5] Shari Lawrence Pfleeger. *Ingeniería de Software*. Prentice Hall and Pearson Education, Enero 2002.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addisson-Wesley, Toronto, Ontario. Canada, November 1994. ISBN 0201633612.

[7] Chuntao Leng and Qixin Cao. Velocity analysis of omnidirectional mobile robot and system implementation. In *Proc. 2nd IEEE International Conference on Automation Science and Engineering CASE '06*, pages 81–86, October 2006. ISBN 1424403103.

[8] Mindstorms. Mindstorms nxt kit. Retrieved April 2008 http://mindstorms.lego.com/Products/Default.aspx.

[9] Rafael Sisto and Santiago Martínez. Campos potenciales. Technical report, Instituto de Computación, Facultad de Ingeniería, Universidad de la República, Uruguay, 2009.